Ilmenau University of Technology
Faculty of Computer Science and Automation
Department of Integrated Communication Systems
Group

Diploma Thesis

Evaluation and Comparison of Service Placement Algorithms in Communication Networks

Inventory number:

Date: 14.03.2011

Author: Imad Kailouh

Student ID: 39174

Date and Place of Birth: Syria, Damascus, 10.12.1976

General Course of Studies: Computer Science Engineering

Major Course of Studies: Integrated Communication Systems

Head Professor: Prof. Dr.-Ing. habil. Andreas Mitschele-Thiel

Scientific Advisor: Dipl.-Inf. Christian Brosch

Abstract

In the 21st century, computers and mobile devices such as laptops, PDAs and mobile phones have become an important part of our daily lives that cannot be dispensed. Nowadays, digitalized information is the faster growing facility used to store and transfer data in the world.

Ubiquitous access to information and services at any moment, any place and in any case at low cost, is one of the main things that should characterize communication networks in the future. These networks, which will interrelate of a variety of systems, should be more dynamic and flexible with regards to changes in access technology, topology, services, etc.

Clearly, one of the most important Challenges that are facing the future networks is the dramatically increased demand for resources or services by the clients. This increment will have negative impacts on the performance of the network, communication cost and quality of service (QoS).

Accordingly, the need arose to match resources supplies with application demands continuously, as these demands are expected to change over time. One of the most commonly methods used to achieve this matching between resources supplies and clients demands is to replicate such resources or services to more than one node in the network.

Creating a replica, gives us the ability to allocate capacity and redirect traffic to accommodate requests and performance requirements. Distributing the content will reduce the latency; Enhances user-perceived performance, and decline the whole network traffic.

In an ideal manner, service's replicas should be located in the network, where a great number of clients are looking for this service. However, in reality it becomes a problem to optimize the service placement.

By using the optimal placement methods for service replicas, we may reduce the used memory and the number of required replicas distributed on the network without losing the quality of performance gained by the employment of these strategies.

This thesis proposes some possible solutions through the implementation and evaluation of four Service Placement Algorithms, with a focus on the evaluation process.

At the beginning, a theoretical study about Service Placement problem is presented. Then, the implementation of the Centralized and Self Organized Service Placement Algorithms in the Common-Simulator environment is followed.

Finally, for proper and ideal evaluation process, it is important to compare and contrast the efficiency and the quality of these different Service Placement Algorithms (Centralized Algorithm based on Cost Function (CAFA), Self-Organized Network Density (SONDe), Self-Organized Random Algorithm based on Probability (SO-RAP) and Self-Organized Advanced Random Algorithm based on Probability (SO-ARAP)).

Zusammenfassung

Im 21. Jahrhundert sind die Computer und die mobilen Geräte wie Laptop, PDA und Handy zu einem wichtigen Bestandteil des täglichen Lebens geworden; auf diese Geräte kann man heutzutage schwer verzichten. Heute sind digitalisierte Informationen das schneller wachsende Medium zum Speichern und Übertragen der Daten in der Welt geworden.

Ein allgegenwärtiger Zugang zu Informationen und Diensten überall, zu jeder Zeit und in jedem Fall zu geringen Kosten, ist einer der grundlegenden Dinge, die das Kommunikationsnetz in Zukunft charakterisieren soll. Diese Netze, die die Heterogenität der verschiedenen Systeme miteinander verbinden werden, diese sollen dynamischer und flexibler in Bezug auf Veränderungen in der Access-Technologie, Topologie, Dienstleistungen, etc. sein.

Die dramatische Erhöhung der Nachfrage nach Ressourcen oder Dienstleistungen durch den Klienten ist selbstverständlich eine der wichtigsten Herausforderungen für das Netzwerk in Zukunft. Dieser Anstieg wirkt sich negativ auf die Leistung von Netzwerk, Kommunikationskosten und Qualität von Service (QoS) aus.

Da erwartet wird, dass die Anforderungen sich im Laufe der Zeit ändern werden, ist die kontinuierliche Anpassung der Ressourcen an die Anforderungen der Anwendungen notwendig.

Eines der am häufigsten verwendeten Methoden, die Übereinstimmung zwischen Ressourcen-und Klienten Anforderungen zu erreichen, ist solche Ressourcen oder Dienste auf mehrere Knoten im Netzwerk zu replizieren.

Das Erstellen einer Replik, gibt uns die Möglichkeit, die Kapazität bereitzustellen und den Datenverkehr so umzuleiten, dass er auf Anfragen und Leistungsanforderungen angepasst wird. Die Verteilung der Inhalte wird die Latenzzeit verringern; verbessert die vom Benutzer wahrgenommene Leistung und verringert den gesamten Datenverkehr.

Idealerweise werden Service Repliken da im Netz platziert, wo eine große Anzahl von Kunden sich befindet, die diesen Service sucht. In Wirklichkeit jedoch wurde die optimale Platzierung der Service Repliken zu einen Problem.

Mit den optimalen Platzierungsmethoden für Service Repliken, können wir die Anzahl der notwendigen Repliken, die im Netzwerk verteilt wurden und die

genutzten Speicher reduzieren, ohne zu viel von dem Gewinn aus dem Einsatz dieser Strategien zu verlieren.

Diese Arbeit schlägt eine mögliche Lösung durch die Umsetzung und Evaluierung von vier Service Placement Algorithmen mit Fokus auf die Evaluierung vor. Am Anfang der Arbeit wird eine theoretische Studie über Service Placement Problem vorgestellt; anschließend folgt die Implementierung der zentralisierten und selbst organisierten Service Placement-Algorithmen in der Common-Simulator Umgebung.

Schließlich ist es wichtig für die richtige und ideale Evaluierung, die Effizienz und die Qualität der verschiedenen Service-Placement-Algorithmen (CAFA Algorithmus, SONDe Algorithmus, SO-RAP Algorithmus, SO-ARAP Algorithmus) miteinander zu vergleichen.

Acknowledgment

I would like to express my gratitude to all those who gave me the possibility to Complete this thesis. First of all I would like to gratefully acknowledge the supervision of my advisor, Dipl.-Inf. Christian Brosch, who has been abundantly helpful and has assisted me in a numerous ways. I specially thank him for his infinite patience, the discussions I had with him were invaluable.

I also would like to say a big thanks to my sister Ola for helping me with the English text and making suggestions for editorial improvements to the Final version of this thesis. I owe my deepest gratitude to my fiancée Eleanor for her support, encouragement, and understanding.

My final words go to my family. I want to thank my Mum and Dad , Azar , Alaa , Etab and John whose love and guidance is with me in whatever I pursue , thank you for making me who I am.

Table of Contents

1 Introduction

1.1 Motivation

Dealing with the problem of choosing which node in a network is the most adequate for hosting a service, which replies to demands from other nodes, can be referred to as a ***Service Placement***. If these services that represent abstractions of collections of applications where placed optimally, it will decrease the network traffic and enhance the connectivity between clients and servers [1].

Nowadays the world is witnessing a rapid growth of a large and a complex communication networks that will face so many challenges and difficulties. One of the most important challenges is the dramatically increased demand for Resources or services by the clients. As a result of this increment, a negative impact will be noticed on the network performance, communication cost and the quality of service (QoS).

Accordingly to that, the need arose to match resources supplies with application demands continuously, as these demands are expected to change over time. One of the most commonly methods used to achieve this matching between resources supplies and clients demands is, to replicate such resources or services to more than one node in the network.

Service replications within a network will give it the ability to enhance its performance, reduce the cost of communication and improve the quality of service. On the other hand, the replication of resources or services does not have many significant positive effects on the network's performance improvements, without choosing the right place where these replications should be located.

The process of defining the suitable nodes in a network that will act as servers i.e.: “Abstractions for locations where services can be hosted”, is described as the ***Service Placement Problem***. All of these makes carrying out a research on this issue very important, valuable, deserves full attention and an effort to develop a powerful Service placement algorithms.

1.2 Problem Statement

Finding the most appropriate nodes, where the services should be placed on, is one of the main obstacles that face the network.

Service Placement Algorithms are developed to help us in finding out solutions for this problem, because of its positive effects on the network performance, communication cost and quality of service.

This paper will provide several solutions to address this problem through (Centralized and Self-organized algorithms). Even so; the solutions presented in this paper attempt to be questionable.

1.3 Goals of this thesis

The major goal of this thesis is to demonstrate the implementation and evaluation of four Service Placement Algorithms with a special focus on the evaluation process. In the beginning, a theoretical study about Service Placement problem is presented. Then, the implementation of the Centralized and Self Organized Service Placement Algorithms in the Common-Simulator environment follows.

Finally, for a proper and ideal evaluation process, it is important to compare and contrast the efficiency and the quality of these different Service Placement Algorithms (Centralized Algorithm, SONDe Algorithm, Random Algorithm and advanced Random Algorithm).

1.4 Structure

The second chapter of this thesis addresses the definition and the clarification of Service Placement Problem, explaining the subject problems and implications. Chapter three gives a theoretical background about the Service Placement Problem, with a mathematical overview for this problem explained by some algorithms, Chapter four deals with the study of algorithms applied in this paper, while (Practical Basis), which is the title of chapter five, discusses the simulation environment, Common-Simulator that will be used in this work.

Chapter six and chapter seven deal with the basic ideas and operations of the Service Placement Algorithms. Chapter six clarifies the implementation of Centralized and Self Organized Service Placement Algorithms in Common-Simulator, all pertinent functions, classes, and procedures. Chapter seven speaks about the evaluation process of implemented algorithms. Finally, the conclusion and the recommendations for future work will be taken up in chapter eight.

2 Service Placement

2.1 Explanation of idioms

Before we define and explain the Service Placement problem, there are some terms should be clarified which can help for a better understanding of this problem. The first question that may come into mind is; what does the word Service mean?

In this thesis the term “service” is used in the same meaning as in the context of “SOAs”:

“A service is a mechanism that enables access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. A service is provided by an entity – the service provider – for use by others, but the eventual consumers of the service may not be known to the service provider and may demonstrate uses of the service beyond the scope originally conceived by the provider. [1]”

A ***Service*** is a “software component executed on one or several nodes of the network. It replies to the service requests received from client nodes through a well-defined interface [2]”. The mutual effect between server and client nodes is referred to by, the function and the protocol of the services that specify the structure, substance and sequence of service demands and replies. In a network, different services may be carried on at the same time, and each node may host more than one service [2].

This comprehensive view on services may be tightened according to the following properties:

- ***Node-specific vs. node-independent:*** A service may be associated with a specific node in the network, e.g. to get an access to special purpose hardware: A service that offers a sensor reading of a certain node in a WSN is node-specific because, the value is based on the physical position of the sensor node [3].

  On the other hand a service may act in an independent way, away from the node it is placed on, e.g. (a cluster head for hierarchical routing). Clearly, Service Placement can only be applied to node-independent services [3].

- ***Centralized vs. distributed vs. Self-organized:*** nowadays, four main methods are in use to deal with the problem of Service Placement. These methods are:
    - Centralized.
    - Distributed.
    - Hybrid (combination of centralized and distributed).
    - Self-organized (an extension of the Distributed-system concept).

Sometimes applying the service may need for it to operate in a centralized manner on one node. Centralized placement techniques have a central unit that gathers information about the whole system. This unit is in charge of managing the placement related functions. Depending on the whole system information hold by the central unit, optimal placement of a service is achieved. A global awareness of service owners, clients, and costs, should be taken into consideration; consequently, they are less scalable and difficult to apply to future communication networks [4].

In our thesis we will discuss an algorithm that is based on centralized mechanism. This algorithm will be characterized by two common ideas:

- Network topology is recognized (well- known).
- Each connection in the network is allocated by the cost.

On the contrary, service may spread itself in the form of a changeable number of equivalent service instances distributed over several nodes [2]. Distributed service placement's strategies, suppose that there is no central unit in charge of managing placement process and each unit possess a part of information about the system status. As a result of this, these strategies will drive to scalability improvement; hence, they can be applied to future communication networks in numerous ways. In addition these strategies make the cost less, improve the scalability but, we can't ensure that the placement of services that we gained is optimal [4].

In this thesis some distributed service placement strategies will be discussed (e.g. SONDe). These strategies suppose that:

- Placement and other related functions are achieved locally.
- No global information about system status (e.g. topology) is needed.

Self-organization: Self-organization can be defined as "a process in which structure and functionality (Pattern) at the global level of a system emerge solely from numerous Interactions among the lower-level components of a system without any external or centralized control.

The system's components interact in a local context either by means of direct communication or environmental observations without reference to the global pattern [5]".

For our goals in the Service Placement Problem, the self-organization term will be understood as the capability of replicating and moving system parts, like service instances, according to changes that come up in the network without the need for central control or human intervention.

- ***Composite vs. monolithic:*** a composite service is given the ability by its flexible structure, to be divided into various interdependent subservices that may operate on more than one node. Each one of these is unique and plays an important role of the overall service. On contrary, monolithic services are not dividable because of implementation issues. Thus, all the software elements all over the nodes will be the same, if a monolithic service is given in a distributed way [3].

  Service Placement can be applied on both monolithic and composite services: For monolithic services, it is able to create any number of analogous instances of the service elements. For composite services, service-specific interdependencies between the subservices, should be noticed and get more attention [3].

Service instances or replicas: we refer to the similar service elements that are performing on several nodes, as service instances. Breeding new instances on different nodes over the network based on client's demands represent the main concept of service distribution [2].

To achieve the coordination among service instances, a swap of information is needed. The expressions "service" and "service instance" may have the same meaning (identical) in case of centralized service [2].

Service demand: is represented by the clients need to use a service operated on specific nodes in the network over a period of time. The main purpose of the whole network is to fulfill the service needs of all clients. The criteria that can tell if a Service Placement Algorithm was succeeded, should take into consideration making the main goal mentioned above accomplished [3].

Network Topology: there are two approaches that define the network topology:

- Physical topology.
- Logical topology.

Physical topology indicates to the organization of devices on the network and how these devices are communicated among them. In contrast, logical network

topology is a term that defines the technique of how the information transfers from one device to another within the network.
Physical and logical network topologies can be represented as a graph (nodes represent vertices - the links represent edges) [3].

Service Configuration: The configuration of a service is a process in which, instances of services are being hosted by a group of nodes. Depending on placement strategy (e.g. network topology) the service configuration is adjusted regularly [2].

During the adjustment process some service instances will vanish, other will be created and some will change their places in order to fill the gap between the current and the optimal configuration.

2.2 Definition of Service Placement Problem

By year 2017, more than 7 billion people will be served by wireless equipments that will grow to more than 7 trillion, depending on the vision of Wireless World Research Forum (WWRF) [6]. As a result of this expected increase in the size of networks, there is a need to reconcile between the available resources and clients requirements, on an ongoing basis as long as these requirements are in change over time [7].

This requires that, the place where the service providers are located on the network, should be adjusted, taking into consideration the present network state without any intervention from human.

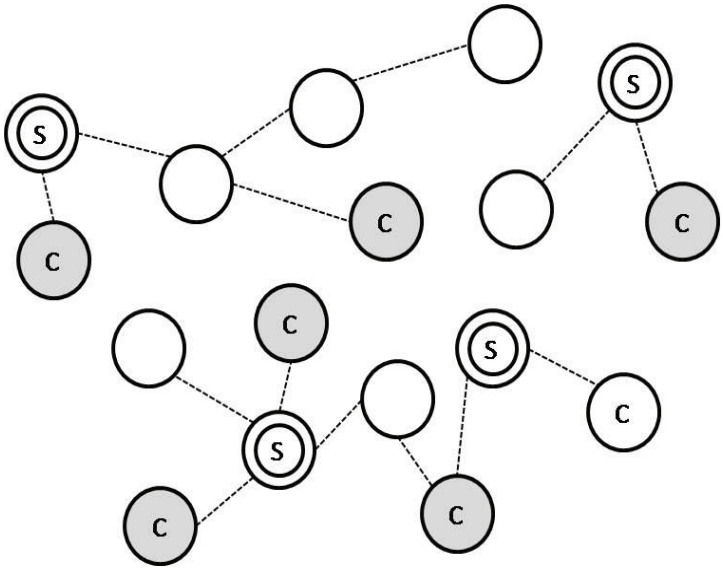

Figure 2-1: Placing of services over the network

Service Placement can be identified as the problem of choosing the most appropriate nodes in the network to be the hosts of the services, which is in charge of responding to the requests from all clients' nodes [2].

Locating the services, that are representing abstractions of collections of applications on the optimal places over the network, will lead to a decrease in the data traffic, and will enhance the connectivity between clients and servers [1].

In other words the problem of service placement can be described as follows:

- A given service to be placed.
- A physical network topology.
- The service requests of the client Nodes.
- Adjusting the location and number of service replicas in the network over time, to ensure that the service request is achieved at a minimal cost, this cost function may include many of parameters for example: (memory usage, transport cost, energy expenditure and CPU usage...etc).

  The service placement policy will be in charge of the selection of the cost function. A placement policy determines the purpose that a service placement system seeks to achieve [7].

  In some service placement algorithms, the process of adjusting where service replicas in the network should be placed on can be achieved, through a non-stopping process of verification that will notice any changes in the network e.g. load increment, server cost rise…etc.

In order to achieve the required adaptation to service requests in a specific area in the network, it is recommended to supply the service by multiple instances; each one of it is placed on a different node "illustrated in figure (2-1)". As the expression Indicates, service instance: is an identical copy of the software element that supplies the service, contains both the executable binary and the application-level data [3].

Every service instance has the ability to supply the whole service on its own. Actually, no difference is noticed between service instances unless for which node of the network they are located on [3].

The difference between distributed and centralized services is expressed by the principle of service instances: practically, it is not allowed for centralized services to generate several instances of the service. Therefore, there is only one service

instance. Actually, this individual service instance of a centralized service is similar to the server in conventional client/server framework [3].

Technically, distributed services have the ability to create several instances of the service, unlike centralized ones. These instances will work together to serve the clients' demands and also can be situated in an independent way. In order to keep the global status of the service synchronized over the whole instances, distributed services will endure an additional overhead (communication overhead) that is not existed for centralized services [3].

The major key questions that should be answered as part of service placement problem solution are as follows [3]:

- Where to locate the service instances in the network?
- How to define the number of service instances needed for optimal operation?
- When to adjust current configuration of services and service instances?
- How to move services and their status from one node to another?

Many algorithms were developed to find the optimal solution to the Service Placement Problem, in other words to answer the above mentioned questions. In our research we will try to answer some of these questions through the implementation and evaluation of four different algorithms.

While studying the four algorithms, the first two key questions will be applied and taken into consideration. Third key question will be implemented only in the second algorithm and, the fourth key question will be applied only in the fourth algorithm.

3 Theoretical Background

3.1 Mathematical Overview (Facility Location Theory)

Mathematically, there are lots of things in common between the service placement problem and the facility location theory, which belong to the field of operations research. This theory tries to find the solution for issues that handle the optimal placement of facilities depending on mathematical models [8].

Among the problems that have been studied in facility location theory, only two were noticed and can be applied to service placement: the k-median problem and the uncapacitated facility location problem [3]. In this section, both of these problems will be explained and the way they can be applied to service placement will be discussed.

3.1.1 The k-median Problem

In a simple way k-median problem can be described as follows:

Let us assume that s= {1... n} is a group represents the possible places where fixed or variable number of facilities (k) should be located. The clients that seek for these facilities are given as a set f = {1... m}. The transportations costs are defined as (n × m) matrix (g_{sf}). The main goal of k-median is to allocate the k facilities at the locations with the purpose of reducing the total cost or meeting the clients' demands. Every customer is served by the nearest available facility.

In the framework of communication networks, the nodes of the network are represented by the vertices (V) and the communication links between these nodes are represented by the edges (E) [3]. According to Hakimi [9] the k-median of graph $G = (V, E)$, can be described in a formal way as a group of vertices $\widehat{Vk}$ like that [3]:

$$\forall^{Vp \subseteq V} \sum_{i=1}^{n} wi \cdot d(vi, \widehat{Vk}) \leq \sum_{i=1}^{n} wi \cdot d(vi, Vk)$$

$vi \in V$ Symbolize the graph's vertex; (wi) symbolize the vertex's (vi) weight, $d(vi, vj)$ represent the weight of the link between two nodes, $d(vi, Vk)$ is the shortest way from vi to its closest component in Vk.

K-median problem can be characterized as follow [3]:

- It's a part of minisum location-allocation problems.
- It works on a discretized field to solve the problem e.g. group of vertex V.
- In a special case, only one vertex in $(\widehat{Vk}, \mathrm{k} = 1)$ that is called the absolute median may exist, we indicate to this problem as 1-median problem.

The k-median problem can be written in a formal way as an integer program.

An integer programming problem: it can be clarified as a mathematical optimization, that a number of its variables or all of them are constricted to be integers [10].

ξij Defined to be an allocation variable likes that [3]:

$$\xi ij = \begin{cases} 1 & \text{if vertex } vj \text{ is allocated to vertex } vi \\ 0 & \text{otherwise} \end{cases}$$

The integer program is to [3]:

$$\text{minimize} \quad Z = \sum_{ij} \mathrm{Wij}.\,\xi\mathrm{ij}$$

$$\text{Subject to} \quad \forall_{\mathrm{j}=1..|V|} \sum_{i=1..|V|} \xi ij = 1 \qquad (3, 1)$$

$$\sum_{i=1..|V|} \xi ij = k \qquad (3, 2)$$

$$\forall \mathrm{i}, \mathrm{j} = 1..\mathrm{n}\ \xi \mathrm{i}, \mathrm{j} \leq \xi \mathrm{i}, \mathrm{i} \qquad (3, 3)$$

$$\xi ij \in \{0,1\} \qquad (3, 4)$$

W is standing for the weight matrix of the distance $\mathrm{Wij} = \mathrm{wi} \cdot \mathrm{Dij}$, and D represents the matrix of the shortest distance $\mathrm{Dij} = [\mathrm{d}\,(\mathrm{vi}, \mathrm{vj})]$ [3].

Each component in the K-element subset is unique and specified to only one vertex; this will be guaranteed through Constraint (3.1). Constraint (3.2) ensures that there are k vertices assigned to them, which Imposes the elements number of the k-median subset to be k. Constraint (3.3) explains that vertices cannot be specified to non k-median vertices. Constraint (3.4) allocates the potential values for ξij [3]. The k-median is $\{\mathrm{vi} \in \mathrm{V} \mid \xi\mathrm{ii} = 1\}$.

3.1.2 The Uncapacitated Facility Location Problem

From an economic point of view, uncapacitated Facility Location Problem can be illustrated as follows:

It's a process that seeks finding the suitable places for facilities e.g. factories or warehouses, to reduce the cost (or to increase profit), to fulfill the demand for some goods. Some costs should be taken into consideration like the cost of facilities establishment and transportation costs (delivering the goods to clients). This issue is referred to as facility location problem [3, 11].

The up-mentioned issue can be exampled in a formal way: a group of clients that seek one kind of goods is represented with (I), and a group of locations for facilities is symbolized with (J).

Constant cost (f_j) will be the result of establishing a facility on locations ($j \in J$). Satisfying the client demands ($i \in I$) from a facility ($j \in J$) will lead to a constant cost (d_{ij}). The way to solve the UFLP, requires establishing facilities on the subset of the existing locations ($\hat{j} \subseteq J$), in order to reduce the whole cost, while making sure that the requests of all clients are met. Every Client ($\boldsymbol{i}$) fulfills its demands from the facility (j) with (d_{ij}) at a low cost [3]. Therefore, the whole cost of an optimal picking of facilities ($\hat{j}$) is represented as follows [3]:

$$\sum_{i\in I}^{n} min_{j\in \hat{\mathbf{j}}}\, dij + \sum_{j\in \hat{\mathbf{j}}} f_j$$

Despite of the similarity in the purposes and techniques, of finding solution between UFLP and k-median problem, it was found that there are three major differences characterize one from the other [3, 12].

- For setting up a facility at a specific vertex ($j \in J$), UFLP will produce a constant cost (fj).

- No restrictions were found on the total number of facilities in UFLP (other than for the clear upper limit |J|) [3].

- On the contrary to k-median problem, which is characterized by a unified group of vertices (V), the UFLP usually use two separated groups, the first one is the group of possible facilities (I) and the second is a group of clients (J) [3].

In a similar way to the k-median problem, the UFLP can be represented in a formal way as an integer program. Let us consider ψ_j and ϕ_{ij} to be allocation variables like that [3]:

$$\Psi_j = \begin{cases} 1 & \text{if facility } j \in J \text{ is open} \\ 0 & \text{otherwise} \end{cases}$$

$$\emptyset ij = \begin{cases} 1 \text{ if the damend of client } i \in I \text{ is satisfied from facility } j \in J \\ 0 \quad otherwise \end{cases}$$

The integer program is to [3]:

$$\text{Minimize } Z = \sum_{i \in I} \sum_{j \in J} d_{ij} . \emptyset_{ij} + \sum_{j \in J} f_j . \psi_j$$

$$\text{Subject to } \forall_{i \in I} \sum_{j \in J} \emptyset_{ij} = 1, \qquad (3, 5)$$

$$\forall_{i \in I, j \in J} \emptyset ij \leq \psi_j, \qquad (3, 6)$$

$$\forall_{i \in I, j \in J} \psi_j, \emptyset ij \in \{1,0\} \qquad (3, 7)$$

Constraint (3.5) ensures that the requests of each client are met. Clients are only served from established facilities; this will be guaranteed through Constraint (3.6). The restriction (3.7) allocates the potential values for ψ_j and ϕ_{ij}. The solution to the UFLP is $\{ j \epsilon J \mid \psi_i = 1 \}$ [3]. UFLP and the k-median problems are NP-hard.

3.1.3 Applicability to the Service Placement Problem

Depending on the definitions of the k-median and uncapacitated facility location problems, it is obvious that there are great similarities with the service placement problem. More specifically, service placement can be considered as an application of facility location theory to service supplying in communication networks [3].

Similarity between service placement problem and the UFPL is clearer than it appears in the k-median problem because; the optimal number of the facilities is depending on the cost of allocating them together with the demands [3].

Even though UFLP and the service placement problem are similar, many differences between of them appear which; can be represented through these three main points:

- As a result of the continuous change of clients' demands over time, the need always arises to adjust the configuration of the services, when the present configuration becomes inappropriate. The idea of changing entries over time is not taken into consideration in UFLP. So it cannot deal with the concept of adjusting a configuration issue [3].

- As a result of the changes that happen to the network topology, service placement is forced to gather the information about the potential places for facilities (J) and the cost that meets the demands of client's (d_{ij}) at the runtime; this leads to a communication overhead that cannot be neglected, while in UFLP this information are available in an easy way without a communication overhead [3].

- The need to swap information among the service replicas, to maintain the global status of the service to operate in unison (synchronization), is not taken into consideration in UFLP [3].

Depending on the differences mentioned earlier, we can't use the same solutions to UFLP for placing services in communication networks. Still, we can form our own algorithms counting on the similarity between these two problems [3].

3.2 Theoretical Overview

In this section some algorithms that provide us with a multiple solutions to the service placement problem will be explained.

3.2.1 BLE-Based Control Algorithm

As the name indicates, this algorithm is built on the idea of the Broadcast of Local Eligibility (BLE) that is applied in the field of robotics, in order to find solutions to a service placement problem. Hierarchical control architecture will be used in this algorithm, to get over the limited scalability problem of the originally suggested idea of BLE [13].

The approach of this algorithm depends on dividing the group of servers into clusters. Each cluster is identified by a unique server called cluster head. Elements of each cluster are able to communicate among themselves by broadcasting messages that can be sent either directly or through the cluster head.

The process of selecting the appropriate place of services in the cluster is achieved through a continuous re-evaluation that depends on adjudication between peer servers. This process is referred to as the decision cycles [13].

In every decision cycle, the following steps will be applied:

- The first step is represented as follow [13]:
  - Every server in the cluster owns its original group of services.
  - It gets new services from other servers in the cluster.
  - Then it broadcasts its original group plus the new arrived services to other servers in the cluster.
  - Finally, the server will host in its list, all services in the cluster.

- While the second step can be explained in the following [13] :
  - Every server will verify how good it can be to host each service.
  - The list of services is classified by each server in accordance with calculated score.
  - The server broadcasts a group of services that are organized according to a specific score, at the same time without going over its capacity.

- Third step: Depending on the comparison between the score list owned by the server, and another one arrived from a peer. The servers will have the ability to know the most qualified one for hosting a specific service [13].

- Fourth step: As a result of the up-mentioned comparison, each server is able to recognize whether to host new services or eliminate the existed ones [13].

  When the decision cycle reaches the end, a comparison process handled by the head of cluster will happen between the primary group of services and the final ones, the rest of services that were not hosted by any server will be exported to the next phase [13].

One of the most important characteristics of this algorithm, that it does not neglect any list of services in the cluster after the decision cycles.
As the servers exchange information among each others, probably a problem will be arise in communication costs regarding the messages number and the time needed to notify all elements of a cluster [13].

3.2.2 Random / Round Robin Load Distribution Algorithm

The most important characteristics of this algorithm are simplicity and statelessness. As the name indicates, this algorithm deals with the Load Distribution problem, either at a random way or depending on the Round-robin scheduling which is one of the simplest scheduling algorithms [13].

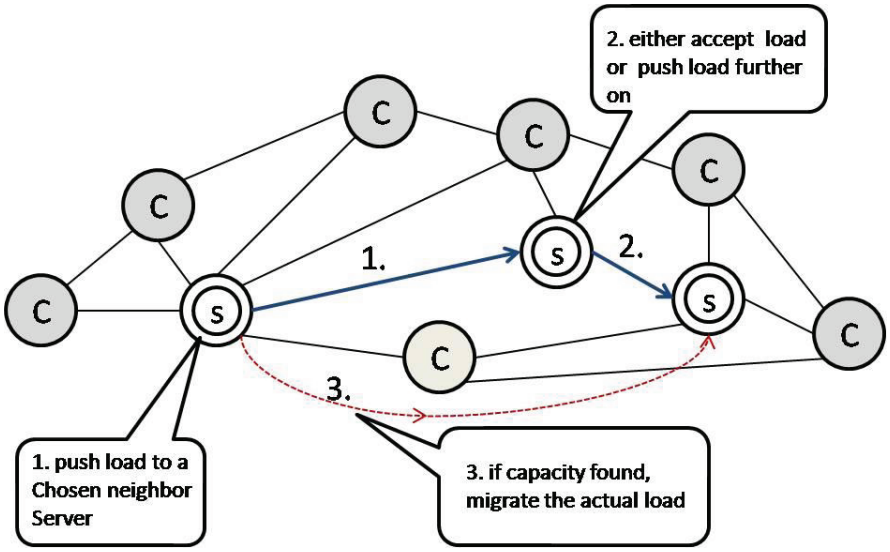

Figure 3-1: Random / Round Robin Load Distribution Algorithms

For example, figure (3-1) shows that when a server suffers from an overload, this algorithm will force the load to move (either randomly or according to the method of Round Robin) to a selected neighbor that may has the ability to endure it if it's possible, or push it away to another server picked in the same way [13].

As soon as a server that has the ability to absorb the load is available, the real movement of the load will be started in the underlying system. Simplicity and statelessness are the advantages possessed by this algorithm, while unpredictability represents the disadvantages. By using a fixed number of hops the termination problem of this algorithm can be solved [13].

3.2.3 Simple Greedy Algorithm

Simple Greedy Algorithm is a simple algorithm belongs to the class of distributed algorithms. The basic principle of this algorithm is based on forcing the load to move to the minimum loaded neighbor.

In contrast to the random algorithms that don't take into consideration any information, greedy algorithms adapt the concept of using the local existing data,

e.g. load situations on the nearby servers. Exchange data among servers is required to make them always informed with the new changes (information) [13].

As mentioned before, this algorithm forces the load to move only towards servers that are not occupied; As a consequence, these servers will rapidly become in use, and the risk of turning into overload situation is increased. This status will may have a negative impact on the system's throughput, to avoid this, greedy algorithms are firmly based on the continuously update of the load circumstances. The problems of termination and cycles can be kept away by using a fixed number of hops. Finding a solution is not ensured in this algorithm [13].

4 Illustrative study of algorithms

This chapter will provide an illustrative study of the four algorithms applied in this Diploma, including the most important characteristics of these algorithms as definitions, system model, problem statement...etc.

4.1 Centralized Algorithm based on Cost function

CACF (Centralized Algorithm based on Cost Function), is a simple centralized algorithm based only on the observation of the cost function of every node in the network. In other words we can say that CACF is a suggested solution to the services placement problem, by selecting nodes that can be called servers or providers, which gives other nodes the service they need.

Expectations: This algorithm will ensures services availability in a centralized way, with a fixed number of service providers related to the whole number of nodes in the network. More precisely, CACF guarantees that a service is located on the nodes that have the best Cost Function values in the network.

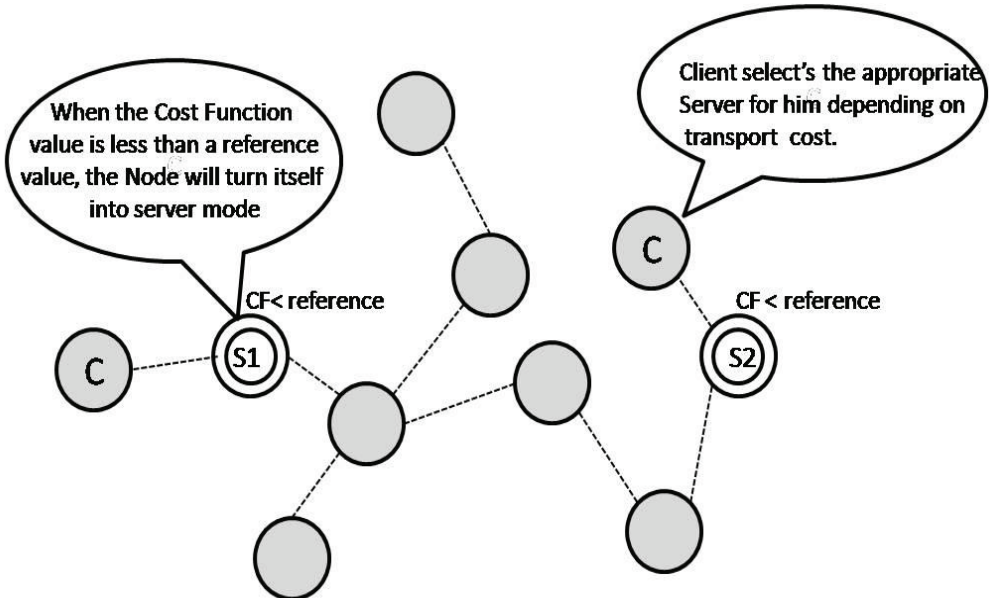

Figure 4-1: CACF with (10) nodes Scenario

The Cost Function is a simple mathematical relationship that expresses the total cost average; or in other words, the average of total resources consumption of every node on the network. Many parameters are involved to achieve the Cost Function; as example the memory cost, CPU cost and others. Cost Function can be represented in the following equation:

$$Cost\ Function = {(CPU\ cost + Memory\ cost + HadrDisk\ cost)}/{n}$$

Where the symbol (*n*) refers to the number of values existed in the numerator.

Transport cost is a parameter that, its value reflects the bandwidth consumption of the link between two nodes in the network. Links between two nodes are asymmetric, which means that the bandwidth consumption values will not be the same in both directions.

In the following, a detailed clarification of the CACF algorithm will be given:

- *System model:* The system consists of a set of (n) nodes, which are connected together. Each node is uniquely identified by its cost function's value, and it has global information about the whole network. Overlay network can be represented as a communication graph, called G.

  Every node in the network can acts as a service provider. Nodes can turn between their two functionalities (client or provider); this switching can be simply achieved. Nodes that have service will be called providers and other will be clients, providers find the way directly to their own service locally. All services over the network are identical.

- *Problem Statement:* the problem addressed by CACF algorithm is:

  - Placing a fixed number of service (x) replications on the nodes that have the most appropriate cost function values in the network.

  - Server's number will be in direct proportion to the whole number of network's nodes.

  - Each client chooses its service provider so that the value of the transport cost between them (the server and the client) is minimal.

- CACF algorithm: At this point, a brief explanation will be given about how this algorithm ensures that service instances will be placed on the appropriate nodes in the network, depending on the cost function value, and how clients are going to communicate with the nearest service provider.

  Figure (4-1) is an example for CACF algorithm, this scenario consist of ten nodes; the algorithm ensures a fixed number of servers that equals one fifth of the total nodes number.

When we start building up the network any of the service instances are not placed yet. Each node will calculate the value of its cost function depending on its private parameters such as CPU cost, memory cost, etc.

In the next step this value will be compared with a reference value, which is constant for all nodes in the network. If cost function is less than the reference value this node will become a server for the rest of the nodes.

Each client searches for and chooses the appropriate server, depending on its calculation of the transport cost between it and the entire service providers that it's connected with. A server that is reached in the minimum transport cost will serve the client. No periodic verification process is performed by CACF algorithm.

4.2 Self-Organizing Network Density Algorithm (SONDe)

SONDe (Self-Organizing Network Density) "is a simple and full decentralized and highly scalable object deployment algorithm for highly requested systems, based only on the Observation of their h-neighborhood [14]". In other words we can say that SONDe is a suggested solution to the services placement problem by choosing nodes referred to as providers, which give the services to other nodes [14].

Expectations: this algorithm will guarantee services existence in the network in a self-organized way, at the same time as reducing the number of service providers. Specifically, this algorithm makes certain that any node in the network will be away from the server equally to a fixed number of hops determined by the algorithm itself [14]. Figure (4-2) shows SONDe Scenario with hops numbers equal to (h=2).

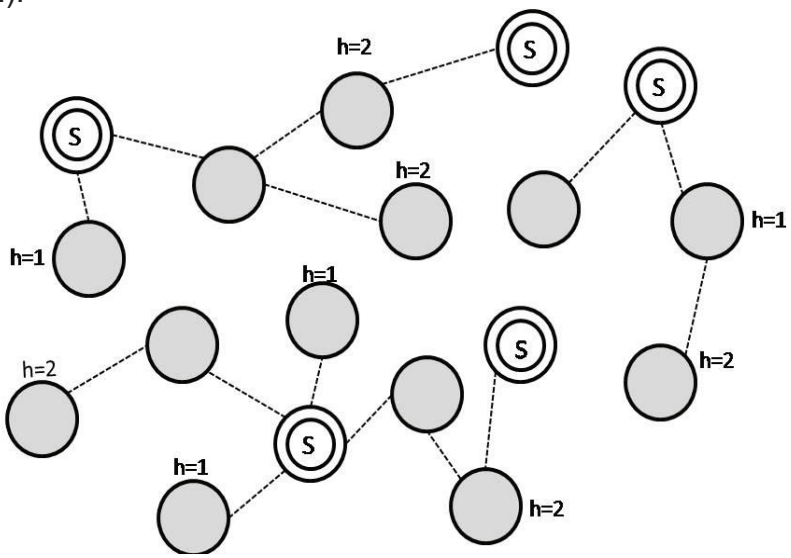

Figure 4-2: SONDe Scenario with hops number equal to (h=2)

Sometimes the network may face a situation in which a few providers are overloaded due to unexpected increase of demands in a given neighborhood, in spite of achieving the service placement criteria; e.g. the availability of the serve in a range of (h) hop for all nodes and the invisibility between the couples of providers.

To get over this problem, certain steps will be taken by every provider, in order to adjust to the overloading circumstances, through taking up advanced search in the overloaded zone.

Consequently, SONDe will be responsible in an independent way for bringing a load balance automatically between the network nodes through a dynamic adjustment of the service provider density. The number of providers will be grown in an automatic way, if a blast of load happened in the network specific zone. On the contrary, the number of service providers in the network will be brought down, in case of load decrease, as a way to save the available resources [14].

Furthermore, SONDe will be self-stabilized, which means that after a period of time the number of providers in the network will be stable.

In the following, a detailed explanation of the SONDe algorithm will be presented:

- *System model:* The system is composed of a group of nodes referred to with (n), which are combined together. Every node is different from the other, and the global information about the whole network is not existed. Overlay network can be exampled as a communication graph, called (G), in which direct neighbors can be defined as a group of nodes that are in contact with node (i) in a direct way [14].

  Commonly, Neighbors which are placed away from a node (i) at distance equal or less than (h) in the network will be represented as a group of nodes ($\mathrm{N}_{\mathrm{i}}^{\mathrm{h}}$), where $(\mathrm{h} \in \mathrm{N}^{+})$ is a fixed value [14].

  The possibility of playing the role of the server or client is available to all nodes in the network. Turning between these two modes can be done in a simple way. Every node that hosts a service will be referred to as a provider and others will be clients. Each provider gets into the service that holds locally [14].

- *Problem Statement:* Reducing the number of replicas of the service to reach a specific limit, at the same time as any node in the network is able to get access to this service within its (h) hops neighborhood. This problem will be explained in SONDe algorithm [14].

There are two main reasons why reducing the number of providers is an important need:

- The process of placing a service instances over the network will consume a lot of resources for the hosted nodes. So, reducing the service number will be considered as a main action that should be taken to keep the resources.

- Placing service instances close to the clients in the network will help, in finding and accessing them quickly.

To describe the up-mentioned statements in a formal way, two familiar definitions from graph theory will be used: independence and dominance.

Independence: The idea of independence can be explained as follow [14]:

- Let us assume that $G = (V, E)$ symbolized a communication graph. Where (V) represents a group of vertex (nodes) and (E) a group of edges (links).

- Let $(I \subseteq V)$ is a sub group of nodes. It can be defined as an independent group, when there is no link in (E) between couple of nodes of (I).
- The reason of using Independence idea is, to represent the concept of constraining the number of providers.

- Resources of whole nodes in the network will not be used totally by service instances, if a group of providers of this service composes an independent group.

Dominance: The idea of dominance can be explained as follow [14]:

- Dominance Concept represents the meaning of availability.

- Let us assume that $\mathrm{G} = (V, E)$ symbolized a communication graph. Where (V) represents a group of vertex (nodes) and (E) a group of edges (links).

- Let $(D \subseteq V)$ is a sub group of nodes. It can be defined as a dominating group when there is a link between any node of $(V \backslash D)$ and a node of (D). To solve this problem (V) must equal (D).
- All clients on the network will have an access to providers of service (s). If these providers formed a dominate group in (G).

SONDe algorithm extends the two previous terms (*Dominance and Independence*), which are only implemented to nodes, (1) hop apart from each other, into nodes that are distant to a value equal or less than h hops, due to the application conditions [14].

Finally, the group that fulfills all the needs of the two previous definitions and referred to as an Independent-dominating group (S) can be defined as follow [14]:

- Let $G = (V, E)$ is a communication graph.
- Let $S \subset V$ is a part of vertices group V.
- S can be defined as an h-independent-dominating group if:

$$\begin{cases} \forall (v, v) \in E^h \Rightarrow v \notin S \ \vee \ v \notin S, \\ \forall v \in V \backslash S \ , \exists v \in S \ s.t. (v, v) \in \ E^h \end{cases}$$

Where E^h represents a group of routes that include no more than (h) sequential links of (E) [14].

- *SONDe algorithm:* two main points will be explained in this section. The first is how exactly SONDe algorithm guarantees that each node in the network will be connected to a server in its neighborhood, while the second one is, how the servers will deal with overload situation [14].

  Every node on the network investigates, if a service provider is existed near to it, the distance between each node and its server must be smaller or equal to the fixed number (h) that represents the hops number.

  The process that is performed by every node to investigate the availability of service providers in its neighborhood will be called the *verification process*. This process will be executed periodically, via message interchange. Communication overhead can be decreased by making servers only reply [14].

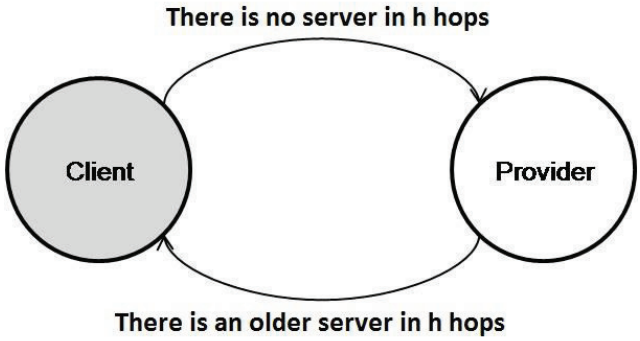

Figure 4-3: Node states (SONDe)

Figure (4-3) shows that every node in the network is identified with two states (client and provider). In other words, each node has the ability to act as a client or as a server depending on some conditions.

Switching between the two previous node's statuses can happen after the periodic verification process, due to:

- If no service provider is available in (N_i^h) node (i) will turns itself into provider.

- In contrast, if node (i) is a provider and has found that another provider is existed in $(\mathrm{N_i^h})$ during its verification process. Accordingly, one of these two providers will stop supplying the service and turn into client [14].

There is a possibility that two nodes belong to the same h-neighborhood may be found at the same time when no provider is available in this region. As a result, both of them will take a decision to act as a provider.

To ensure that the number of service providers in h-neighborhood is limited SONDe algorithm takes the following steps:

- Every node will save the time while switching from its client status to its server status. This time will be called the age of the provider and it will be combined with the node (ID) [14].

- Every provider send its age as a part of its reply messages, and when this message reaches another provider that is existed in the same area, the one who has the oldest age will keep acting as a provider, as for the youngest, it will turn into a client [14].

SONDe algorithm guarantees the availability of one provider in each h-neighborhood. By a dramatic increase of the clients number that may consume the provider resources totally; Some servers maybe exposed to a blast of load, that will lead to a negative influence on the quality of service as a result of (responses and requests) disorder [14].

To deal with the load problem as shown in Figure (4-4), SONDe algorithm adapts a mechanism that takes into account the simplicity and local processing to make the number of servers grow in the region that will be influenced by the overload [14].

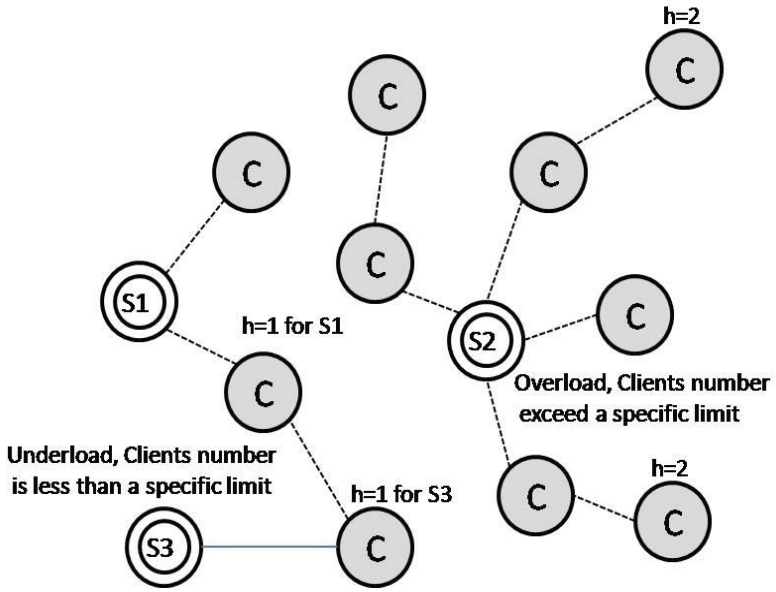

Figure 4-4: Overloads, Under-load Situations (SONDe)

In contrast, if a provider suffers from under load, which means that the number of clients it serves is less than a specific limit for a particular period of time. SONDe will adapt a mechanism that takes into consideration the simplicity and local processing to make the number of providers decreased in the area that will be influenced by this under load [14].

As mentioned before, SONDe algorithm is characterized by a process called the periodic verification process; that is performed from each node to observe any changes over the network such as overloaded situation. SONDe algorithm is a self-stabilized, which means that the number of providers will be stable and will never change dramatically after a period of time.

4.3 Probability Random Algorithm (Self-Organized)

SO-RAP (Self-Organizing Random Algorithm based on Probability) is a simple Self-Organized algorithm, based only on the observation of the random value, which is generated randomly by every node in the network. In other words we

can say that SO-RAP is a suggested solution to the service placement problem by selecting nodes, also called servers or providers, which holds the service.

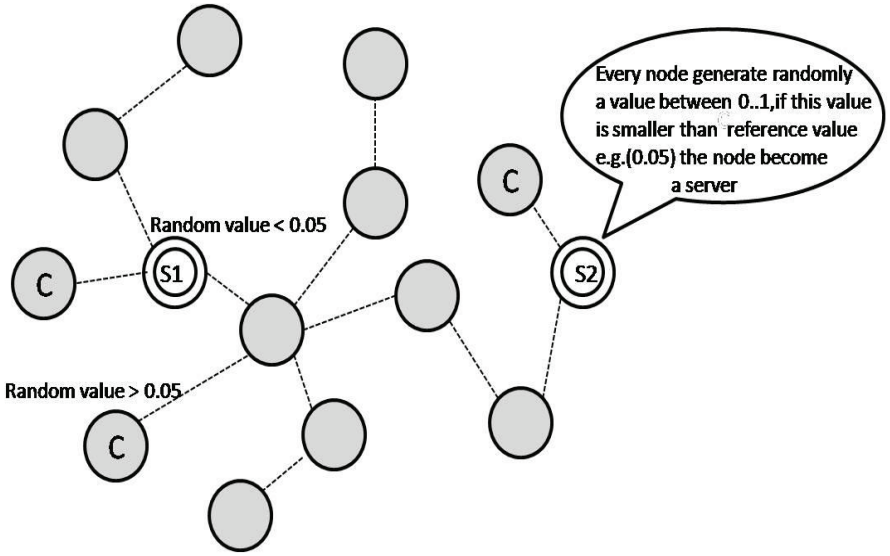

Figure 4-5: Placing service instances over the network (SO-RAP)

Expectations: what is expected from this algorithm is placing a random number of service instances on different nodes in the network.

An illustration of the SO-RAP algorithm will be provided, as follows:

- *System model:* The system is a collection of nodes (n), which are connected together. Every node is characterized by its random value and any global information about the whole network is not possessed. Overlay network can be represented as a communication graph, referred to as G.

  Every node in the network is identified with two statuses that can switch among them (the client state and the provider state). Switching between these two statuses can be performed in a simple way; Providers get into their own service locally. All services over the network are identical.

- *Problem Statement:* placing a number of service x replications on the nodes over the network in a random way, will be the issue addressed by SO-RAP algorithm.

- SO-RAP algorithm: At this point a brief explication will be given about how this algorithm works in order to place service instances over the network.

  Figure (4-5) is an example for SO-RAP algorithm. First, none of the service instances are placed yet on the network, each node will generate randomly a value within a range between 0 and 1.

All nodes in the network hold a fixed value as a reference e.g. (0.05); the only criterion that should be used in order to select this value is to be useful, so that we get the number of servers commensurate to the total number of nodes of the network, as example it is not practical to choose a reference value that equals to (1) thus, all nodes will become servers.

Independently, each node performs a comparison between the value that is randomly generated and the reference value.

If the random value is smaller than the reference value, then this node will become a server for the rest nodes of the network. The number of service providers will be random at every time this algorithm is implemented on the network.

As a drawback of this algorithm, sometimes any node in the network may not be appropriate to host the service, because all of the nodes may have a random value greater than the reference value. No periodic verification process has been used in this algorithm as SONDe. This algorithm will be the basis for the next algorithm in our research.

4.4 Advanced Probability Random Algorithm (Self-Organized)

SO-ARAP (Self-Organized Advanced Random Algorithm based on Probability) is a development of the (SO-RAP) algorithm. It is a Self-Organized algorithm based on the monitoring of a value generated randomly by every node in the network and also, on the observation of the h-neighborhood for each server.

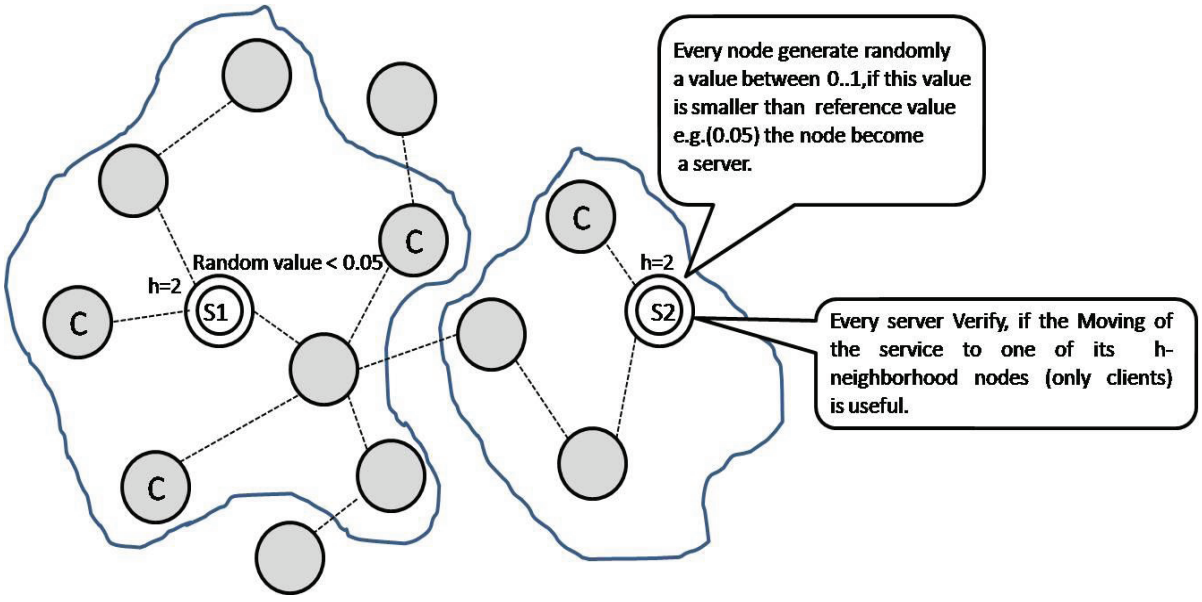

Figure 4-6: Placing of service instances over the network (SO-ARAP)

In another way we can say that SO-ARAP is a suggested solution to the services placement problem by picking out nodes, also named servers or providers that control the service.

Expectations: What is expected of this algorithm can be clarified, through the following two points:

- Placing a random number of service instances on different nodes in the network.
- Transfer the service instances to the appropriate node within its h-neighborhood.

The following represents a detailed description of the SO-ARAP algorithm:

- *System model:* The system can be described as a group of nodes (n) that are communicating with each other through links (e.g. P2P links, wireless links...). Every node is uniquely identified by its random value and it does not have any global knowledge about the entire network. Overlay network can be illustrated, as a communication graph, named G.

  Two different situations define the functionality of each node in the network, the client and the provider. Changing between these two statuses can be simply done; providers access in a direct manner to their own service locally. All services over the network are the same.

- Problem Statement: placing a number of service x replicas on the nodes over the network in a random way, then verifying if the movement of these services within their h-neighborhood is useful.

  Each server verifies for its neighborhood periodically (within h hops): the costs for server on the current node, and the costs for server if one of the neighboring nodes becomes the server instead of the current node.

  If the provision of the server is cheaper on a neighboring node, then the service is shifted there. This will be the question addressed by SO-ARAP algorithm.

- SO-ARAP algorithm: At this point a short clarification will be provided about how this algorithm acts in order to place service instances over the network.

  Figure (4-6) is an example for SO-ARAP algorithm. In the beginning, any of the service instances are not located yet on the network. A value within a range between 0 and 1 will be generated randomly by each node. All the nodes on the network own a constant value as a reference e.g. (0.05).

Independently, each node performs a comparison between the value that is randomly generated and the reference value. As a result, if the random value is smaller than the reference value, this node will become a server.

Subsequently, every server verifies if the movement of the service to another node within its h-neighborhood is valuable. The periodic verification process will include the following steps:

At first, every server calculates its cost function on its current node, and then all clients served by this server will also calculate their cost function.

In the next step, transport cost between clients and their server will be computed. Each client computes the transport cost between it and all other clients within h-neighborhood together with the server.

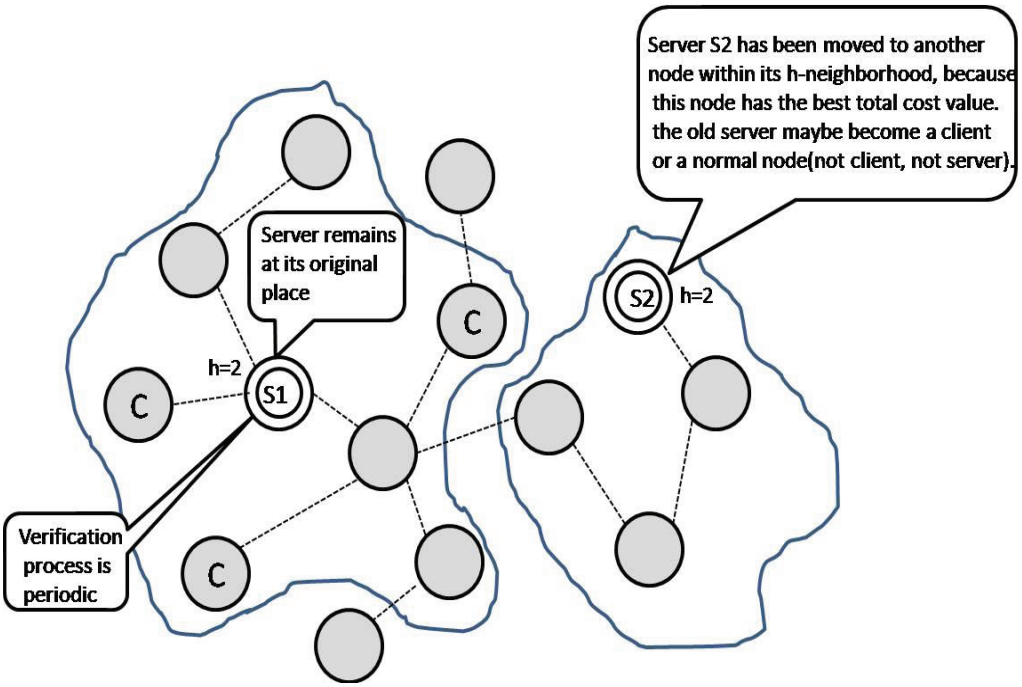

Figure 4-7: The movement of the service to another place (SO-ARAP)

The total cost for each node (client or server) is a result of, the addition of the cost function to the transport cost. The Node with the best value (smaller total cost value) will be a candidate to become the server.

Figure (4-7) shows that the server S2 has been moved to another node (client), because the new node has the best total cost value than the old server, while server S1 remains at its original place. The verification process is periodic, that's mean it will be repeated after a fixed time value, and this will ensures that the service will be placed In accordance with changes in the network.

As a result of that, the movement of the server probably makes sense:

- If the server gets closer to the clients (lower transport costs).
- If the cost function (CPU and memory costs) of the suggested server is small (e.g. because there are more resources available).

5 Practical Basis (Simulator)

This chapter will provide a detailed explanation about CommonSim Simulator, which is used to implement our service placement algorithms that will be studied in this thesis.

5.1 CommonSim

CommonSim; is a java-based Simulator. It can be defined as follow: “a discrete event simulator framework, which is a flexible solution for discrete event simulator needs; that aims to provide a low entry barrier as well as easy extensibility” [15].

The CommonSim simulator framework mainly depends on a software called "Apache Maven” that manage its building and operating process. Depending on that, a certain installation step is not required.

CommonSim will be managed by Maven in accordance with the information provided in the simulation Project Object Model (POM) [15].

During the installation of Apache Maven, and while compiling the first simulation, the simulator setup will be accomplished. One of CommonSim advantages is the ability to install multiple versions of simulator framework without them interrupting with each other [15].

In the following, a brief explanation will be provided, regarding the usage of the CommonSim discrete event simulator framework. Giving assistant with the framework’s structure that help for more understanding of JavaDoc API, and providing the ability to achieve the required simulations will be the major purpose of this explanation [15].

- The issue of compiling a simulation as an initial step is indeed the major thing needed to establish the CommonSim environment. Apache Maven is applied to manage the Simulator software [15].

- By compiling a simulation for the first time, all required dependencies will be downloaded and installed by Maven, together with the simulator itself. The installation of the JDK and Apache Maven are very necessary to build the simulation environment [15].
  Note: the installation of the CommonSim simulator is possible without any need to Maven.

- A particular maven archetype will be used in order to set up the simulation project structure. This can be done by typing the following instructions in the console [15]:

  "mvn archetype: generate -DarchetypeCatalog=http://ih55.theoinf.tu-ilmenau.de/"

- An appropriate archetype must be chosen from the given list that corresponds to the type of project we want to build. The user should enter some information, important to maven such as (artifactId, package names etc...) [15].

- When the generation process is completed, a new directory will be created with a name similar to the given artifactId. Figure (5-1) illustrates what this directory contains. The values, which are given between (<>) represent the user entries. All essential dependencies and setup commands are included in the pom.xml file [15].

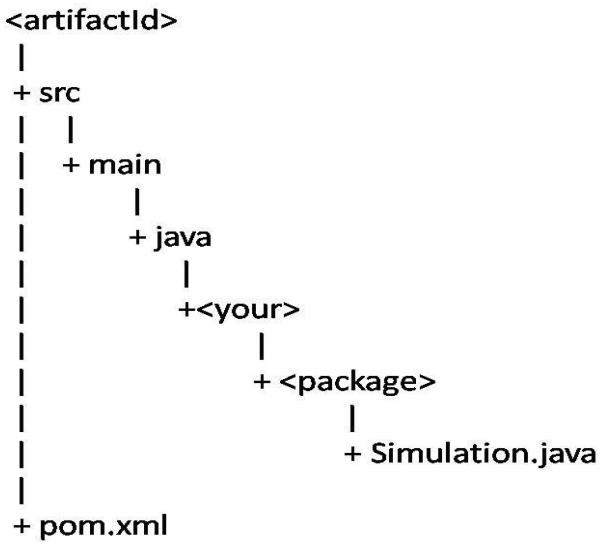

Figure 5-1: Directory structure in CommonSim

- Event: is the most important entity of execution in "CommonSim" the discrete event simulator. "Everything that happens inside the simulation is implemented as a series of events that occur at specific times. The events are scheduled by the simulator kernel [15]".

- "Simulation.java" file contains the Simulation class as the following code shows.

```
package <your>.<package>;
import de.tuilmenau.ics.CommonSim.model.ISimulation;
/**
 * Simulation
*/
public  class  Simulation implements  ISimulation
  {
   @Override
    public  void setup()
     {
       // TODO Auto-generated. Put in things to create your model.
     }
    @Override
     public  void  tearDown()
      {
        // TODO Auto-generated. Put in things here to clean up.
      }
  }
```

Simulation class has two methods, the setup and the teardown. Setup method is in charge to build all the objects needed to start the simulation, while teardown method is responsible for demolishing all objects when the simulation process is finished. As the previous code shows there are no events applied yet, as a result the simulation will be accomplished without any event being planned [15].

- As an example of CommonSim event implementation, a new file called PrintEvent.java with the following code will be created:

```
package <your>.<package>;
import  de.tuilmenau.ics.CommonSim.core.IEvent;
import  de.tuilmenau.ics.CommonSim.core.Environment

public  class  PrintEvent  implements  IEvent
{
   @Override
    public  void  fire()
    {
      System.out.println("Event  got scheduled at:"
                                +Environment.get().getKernel().
                                  getCurrentTime().toString());

    }

}
```

And then modifications in the “Simulation.java” file will be as follow:

```
package <your>.<package>;
import de.tuilmenau.ics.CommonSim.model.ISimulation;
import de.tuilmenau.ics.CommonSim.core.Environment;

/**
* Simulation
*/
public  class  Simulation implements  ISimulation
  {
     @Override
      public  void setup()
        {
            Environment.get().getKernel().scheduleAt(1.0, new
            PrintEvent());
        }
      @Override
       public  void  tearDown()
        {
         // TODO Auto-generated. Put in things here to clean up.
        }
   }
```

As mentioned before, a Simulator needs to be downloaded at the first time we run and compile the simulation. This procedure requires being connected to the internet and, it may take some time. As a result of running and compiling the previous simulation example, some outputs will appear on the console about initialization of the framework, choosing of the kernel etc [15].

When the statement “Starting simulation” appears on the console something like "Event got scheduled at: 0:00:1, 000000000000" should be seen. This indicates that the installation of the simulator was done successfully [15].

5.2 Conclusion

CommonSim Simulator was used while implementing our algorithms. This program is Easy to deal with and it’s a flexible solution for discrete event simulator needs. It has been developed by the Integrated Communication Systems Group, at TU Ilmenau. For all reasons mentioned above CommonSim Simulator was chosen to be the evaluation tool for our algorithms in this thesis.

6 Implementation

This chapter provides an explanation of the most important implementation points, to ease the transition to the examination of the source code for the four Service Placement Algorithms studied in this thesis. As implementation tool, CommonSim Simulator was used, which is based on Java Programming Language.

6.1 Important Implementation Aspects (CACF)

This section will discuss the basic idea of the Centralized Algorithm based on cost function, and provides a clarification of some classes and functions that form its structure.

6.1.1 Basic concept (CACF)

1. The basic idea of this algorithm is based on, the fact that each node in the network has the ability to compute a value of a specific function called the cost function (CF) mathematically. This value reflects the status of this node depending on the availability of its resources. Every node in the network is identified by two modes: Client and Server.

2. Cost Function value represents the percentage of arithmetic average of the following parameters: CPU, Memory and Hard Disk Usage. These Parameters are real values within a range of (1...100), e.g. (CPU Cost=34.6, Memory Cost = 67.68, Hard Disk Cost = 12.4, Cost Function = (34.6+67.68+12.4)/ (3*100) = 0.38).

3. The calculated value of the Cost Function of each node will be compared with the reference value e.g. (RF = 0.3), which is the same for all nodes on the network. If (CF < RF) then the node will turn itself into the Server mode, if not it will remains as Client. In our example mentioned above, the node will act as a Client because (0.38 > 0.3).

4. This algorithm ensures the availability of a fixed number of servers, proportional with the total number of nodes in the network. If the number of servers resulted from the application of this algorithm is more or less than the desired number, this algorithm will readjust the reference value (RF) until we get the appropriate number of servers.

5. Criterion upon which, the appropriate server for each client in this algorithm is selected, is based on the calculation of the transport cost between this client and the servers associated with it. Server that can be reached at the lowest transport cost will be chosen. This will be represented in figure (6-1).

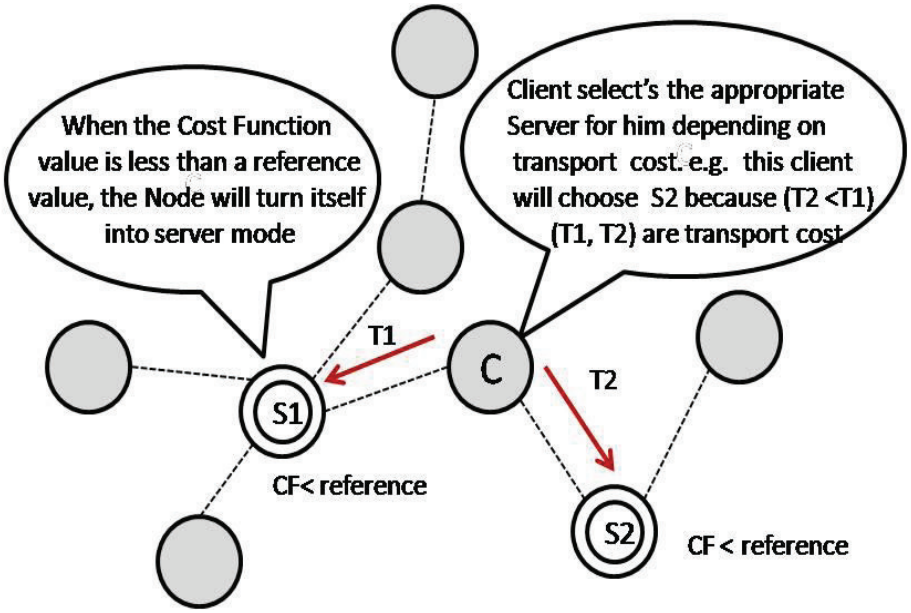

Figure 6-1: Selecting of the server based on transport cost (CACF)

6.1.2 Structure of CACF

The structure of this CACF algorithm consists of two files the (*Node.java*) and (*StartSimulation.java*). All methods and classes definitions are contained in these two files.

In the following, some important functions of the source code will be explained.

In Node.java:

CostFunction: This function is responsible for calculating the percentage of the arithmetic average of the following parameters: CPU, memory and hard disk usage.

SrOrCl: Depending on the result of the comparison between the value of the cost function and the reference value, this function will take a decision to shift the node functionality either from a client mode into a server mode, or to stay acting as a client.

In StartSimulation.java:

SearchForSer: The main task for this function is to search for servers that the client can communicate with, in addition to calculate the transport cost between it and each one of these servers.

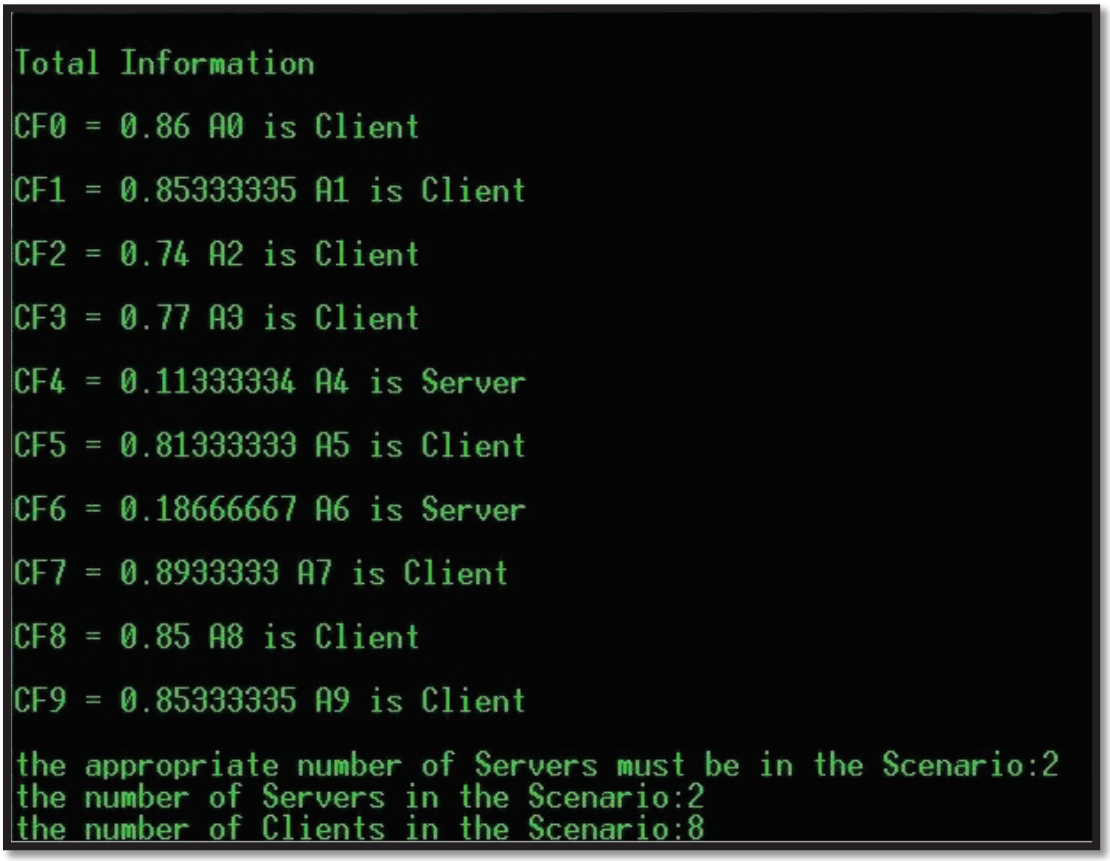

Figure 6-2: Screenshot of simulation results of CACF algorithm

The screenshot as shown in figure (6-2) shows the result of a simulation process of ten nodes scenario; in which fifth of the total number of nodes will become servers. It's clear that two nodes become a server (A4, A6) because their CF values are smaller than the reference value, which is equal to (0.3). Other nodes will act as clients and each one of them will choose its appropriate service provider from these two, depending on the transport cost between each client and these servers.

6.2 Important Implementation Aspects (SONDe)

This section will explain the basic idea of the Self-Organized Network Density algorithm (SONDe), and gives an explanation of some classes and functions that form its structure.

6.2.1 Basic concept (SONDe)

SONDe is a self-organized service placement algorithm that is based on the principle: (each node in the network monitors all of its neighbors, which are at a distance less than or equal to h-hop, periodically and asynchronously).

In addition to the (h) parameter, each node will be identified also with another parameter called (h-local ≤ h), which is generated randomly and periodically updated.

The following will explain the steps to be undertaken by each node independently. Further below stands, what will comes up, when these nodes interact with each other.

What does each node do?

All these operations are performed regularly, e.g. every 10 seconds

1. Monitoring process tends to verify the server existence or non-existence for each node within its h-neighborhood. When a server does not exist and the node is a client, this node will turn itself acting as a server.

2. On contrary, if the node is a server and another server was existed this will lead that; one of them should turn itself into a client depending on some conditions.

   The reason why two servers can be found inside the same area will be clarified as follows: when two nodes in the same h-neighborhood start their monitoring process simultaneously. Consequently, both of them will find that no server is available, as a result of this they turn to act as servers.

   As an attempt to reduce the number of servers while maintaining their availability, this algorithm starts to record the time of the transition from client to server status, which will be called the server age.

Depending on the server age and the (h-local), a comparison between servers located in the same area can be raised. If the two servers have the same (h-local), then the server with the youngest age will stop acting as a server and turn into a client.

3. If (h-local) for a certain period, for example 10 cycles, has not changed. The node will compute the average of all (h-local) values of its direct neighbors and set this value to (h-local).

4. If the node is a server:
    - If the server is overloaded for a while e.g. 5 cycles and its (h-local) still greater than 0, this leads to: the server's (h-local) will be reduced by 1, as for all direct neighbors nodes their (h-local) will be set to the server's (h-local) new value.

    - If the server is under loaded for a while e.g. 5 cycles and its ((h-local) < h), this leads to: the server's (h-local) will be increased by 1, and for all direct neighbors nodes their (h-local) will be set to the server's (h-local) new value.

What happens when all nodes interact with each other?

1. Some nodes in the network become servers.
2. The rest of the network nodes are clients, each one will register itself in a specific server.
3. A Server is overloaded:
    - It reduces its (h-local) by 1 and all its direct neighbors also get the same value of the new (h-local).

    - All other nodes in the network update their (h-local) after a specific period of time, by taking the average of its direct neighbors (h-local). The decrease of (h-local) values around the overloaded Servers from previous step will be propagated so on in the area.

    - Due to the decrease of values of (h-local) in the vicinity of the overloaded server, some clients will no longer find the server within their (h-local range); as a result new servers will be created.

    - The clients, which are now served by the new server, will never cause any load on the original server, then the load of this server is

reduced. If the server becomes overloaded after a period of time, the same process from the first step is performed again.

4. The under loaded case of a server is similar to the overload one, except that (h-local) is increased by 1 (maximum up to h). This will usually disable some servers in the neighborhood of the server and then the low-utilized servers will get more clients.

6.2.2 Structure of SONDe

The structure of SONDe algorithm consists of two files the (*Node_1.java*) and (*StartSimulation_1.java*). All methods and classes definitions are contained in these two files.

In the following, some significant functions of the source code will be clarified:

SearchForServer: The main task of this function, which is executed by every node in the network, is to search for a server within the node h-neighborhood, where (h) is a fixed number of hops. If there is no server existed, then the node will turn itself to act as a server.

SearchForOldServer: This function is performed by the node, when it acts as a server. Its main task is to find out if another server within the h-neighborhood exists.

If there is another server, this function makes sure that the two servers possess the same h-local value, and then it raises a comparison between the server's ages. The server with the youngest age becomes a client.

OverLoad: this function is referred to, when the server suffers from overloading. In other words that means; the number of clients served by this server is increased exceeding a specific limit.

UnderLoad: this function is indicated to, when the server suffers from under loading. In other words that means; the number of clients served by this server is decreased exceeding a specific limit.

getNbrAvg: This function is responsible for the calculation of (h-local) value, if this value does not change after a certain period of time, the node will call this function to compute the average of all (h-local) values of its direct neighbors, and then set this value to the (h-local) of this node.

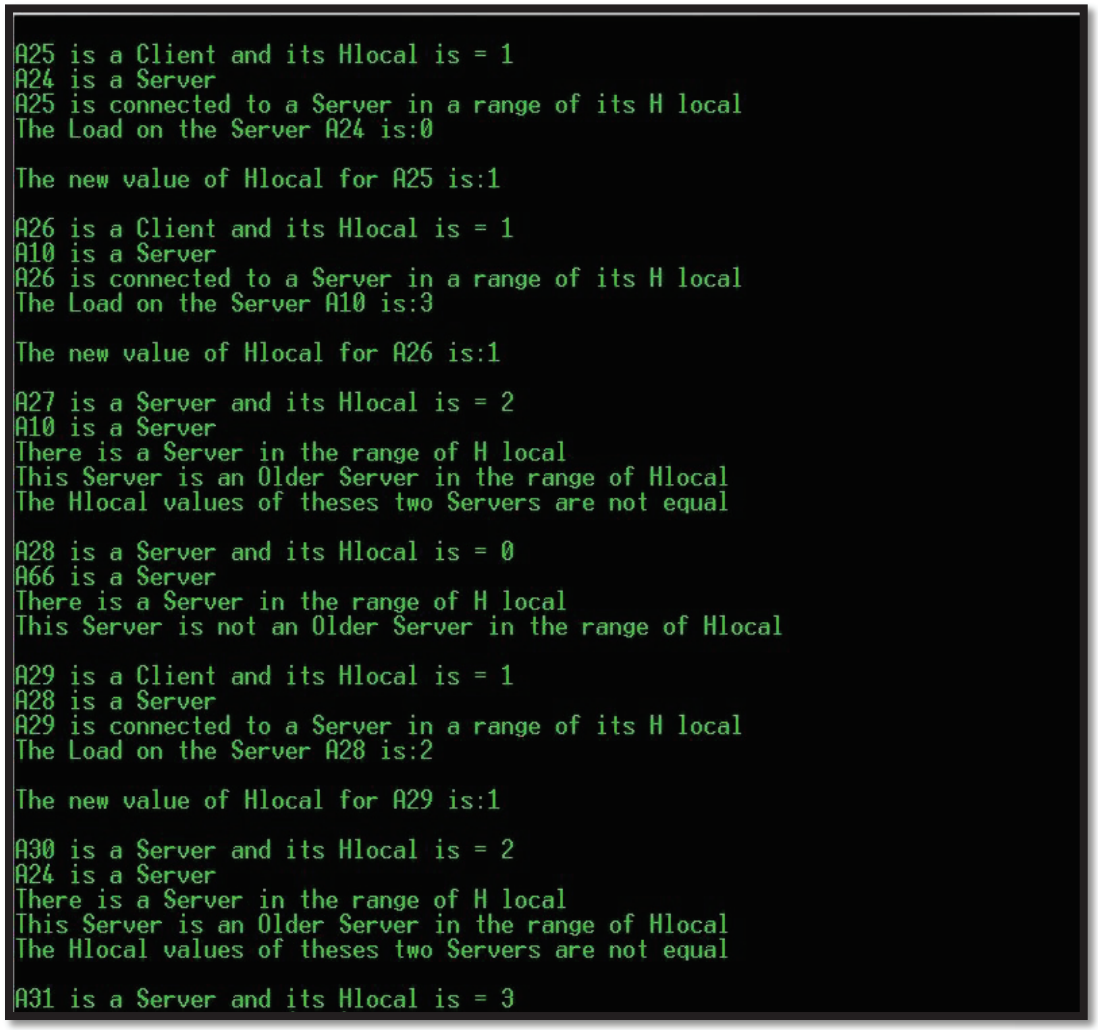

Figure 6-3: Screenshot of simulation results of SONDe algorithm

The screenshot as shown in figure (6-3) represents a part of the simulation results of (100) nodes scenario. It's obvious that each node is identified with a parameter called (Hlocal < h) generated randomly e.g. (Hlocal for A25 is equal to 1). This parameter will be changed depending on some conditions during the running of the program. Every client will be connected to a server e.g. clients (A25, A26, A29).

When a server detects that another server is in its h-neighborhood, it will investigate if this server is older and has the same (Hlocal). When these two conditions are achieved the server will turn itself into a client status e.g. (server A27 has found another one in its h-neighborhood, but their Hlocal values were not the same).

The result shows the load that was applied on each server during the running time e.g. (server A10 has a load equals to 3 clients). Load should not exceed a specific limit.

6.3 Important Implementation Aspects (SO-RAP)

This section describes the basic concept of the Self-Organized Random Algorithm based on Probability (SO-RAP), and provides an illustration of some classes and functions that form its structure.

6.3.1 Basic concept (SO-RAP)

Condition:

A ratio specified which portion of the nodes of the network are to become servers. For example (5%).This means that approximately 5% of the nodes in the network will become servers, the server nodes are coincidentally selected. This ratio is well-known to all nodes of the network.

Procedure:

Each node generates a random number between 0 and 1. The interval of the random numbers will be divided according to the above rate.
At 5%, this means:

- Random number between 0 and 0.05 -> node is a server.
- Random number 0.05 to 1.00 -> node is not a server.
- (0.05 is 5% of 1).

Example:
Network with 5 nodes, ratio = 5%=0.05, (Random number > 0.05) -> not a server, (Random number < 0.05) -> Server.

Node 1: A generated random number = 0.30 -> not a server.
Node 2: A generated random number = 0.01 -> Server.
Node 3: A generated random number = 0.75 -> not a server.
Node 4: A generated random number = 0.69 -> not a server.
Node 5: A generated random number = 0.03 -> Server.

In Addition, we may face a situation in which more or less than 5% of the nodes can become servers.

6.3.2 Structure of SO-RAP

The structure of SO-RAP algorithm consists of two files the (*Node_2.java*) and (*StartSimulation_2.java*). All methods and classes definitions are included in these two files.

Randomvalue: Depending on this function, a random value between (0...1) is generated for each node in the network.

```
A15 is a Client
A23 is a Server
A13 is a Client
A8 is a Client
A12 is a Client
A28 is a Client
A18 is a Client
A29 is a Client
A19 is a Server
A22 is a Client
A24 is a Client
A27 is a Client
A3 is a Client
A11 is a Client
A2 is a Client
A5 is a Client
A0 is a Client
A21 is a Client
A26 is a Client
A16 is a Client
A10 is a Client
A4 is a Client
A14 is a Client
A25 is a Client
A9 is a Client
A20 is a Client
A7 is a Client
A17 is a Server
A1 is a Client
A6 is a Client
9276 [de.tuilmenau.ics.CommonSim.MainApplication.main()] INFO de.tuilmenau.ics.C
ommonSim.controller.DefaultController  - Simulation finished.
9282 [de.tuilmenau.ics.CommonSim.MainApplication.main()] INFO de.tuilmenau.ics.C
ommonSim.controller.DefaultController  - Fired 1 events in 9181 milliseconds (0.
10892059688487093 events/second)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 17 seconds
```

Figure 6-4: Screenshot of simulation results of SO-RAP algorithm

The screenshot as shown in figure (6-4) represents the simulation results of the (30) node scenario. Each node generates a random value between (0...1), if this value is smaller than a reference value, the node turns its functionality into a server .e.g. (A23, A19, A17) are the nodes that will act as servers in the network as the screenshot (6-4) shows.

6.4 Important Implementation Aspects (SO-ARAP)

This section gives details about the basic concept of the Self-Organized Advanced Random Algorithm based on Probability (SO-ARAP), and provides an explanation of some classes and functions, which built its structure.

6.4.1 Basic concept (SO-ARAP)

1. Random placement of the servers depending on the previous algorithm (SO-RAP) should be available.
2. Verifying that the movement of the servers is useful, each server checks for its neighborhood (within h Hops):

    - The costs for server on the current Node.
    - The costs for server, if one of the neighbor's nodes instead of the current node will become a server.

If the establishment of the server on a neighbor's nodes is cheaper, the server will be shifted there. This examination is carried out regularly by every server individually. Besides, the number of the servers on the network remains steady.

The costs to be considered in step 2: (CPU, Memory, Transport (between client And server)).

Shifting the server to its new location probably does make sense:

- If the server comes closer to the clients (transport costs decrease).
- If, on potential nodes that are suggested to act as a server, the CPU and the Memory costs will be less (this means more resources available).

6.4.2 Structure of SO-ARAP

The structure of SO-ARAP algorithm includes two files the (*Node_3.java*) and (*StartSimulation_3.java*). All classes and methods definitions are contained in these two files.

Searchfor: The main objective of this function is to identify clients, which are dealing with each server. Each client registers itself only with one Server**.**

SerachForSer: This function is responsible for determining, if the node is the server or not. In addition it calculates the cost of transportation of this server, which represents the sum of costs of transport between each client and the server.

SerachForCl: The primary task of this function is to identify the nodes that represent the clients to a specific server, and then calculate the cost of transport for each client separately. Client`s transport cost is represented as the sum of the transport costs between this client and the other clients (Server node can be a client or not).

Figure 6-5: Screenshot of simulation results of SO-ARAP algorithm

The screenshot as shown in figure (6-5) demonstrate a part of the simulation results of (50) node scenario. First of all, service instances are placed over the network in a random way based on SO-RAP algorithm.

Depending on the resources consumption such as memory cost, CPU cost, transport cost, this algorithm makes the decision to move the service from its current place to another within the h-neighborhood, So that this transition is useful e.g. server (A18) serves two clients (A6, A8), the comparison between the total cost for the server and these two clients will decide if the shifting of the service from A18 to A6 or A8 is meaningful. A18 has a total cost value equals to (295), total cost for A8 is (174) and for A6 (263). As a result, the service will move to A8 and A18 turning into a client.

7 Evaluation

In overview, this evaluation provides a description of the implemented scenarios for each algorithm (CACF, SONDe, SO-RAP, and SO-ARAP). Then there is an analysis of the graphs, followed by a comparison.

7.1 CACF Evaluation

Common-Sim simulator is used to evaluate the performance of the CACF (Centralized Algorithm based on Cost Function).the number of nodes in the evaluated scenarios is as follow (10, 20, 40, 50, 100, and 200). Nodes in the network are not mobile and they are connected in a P2P manner (not all nodes are connected together).

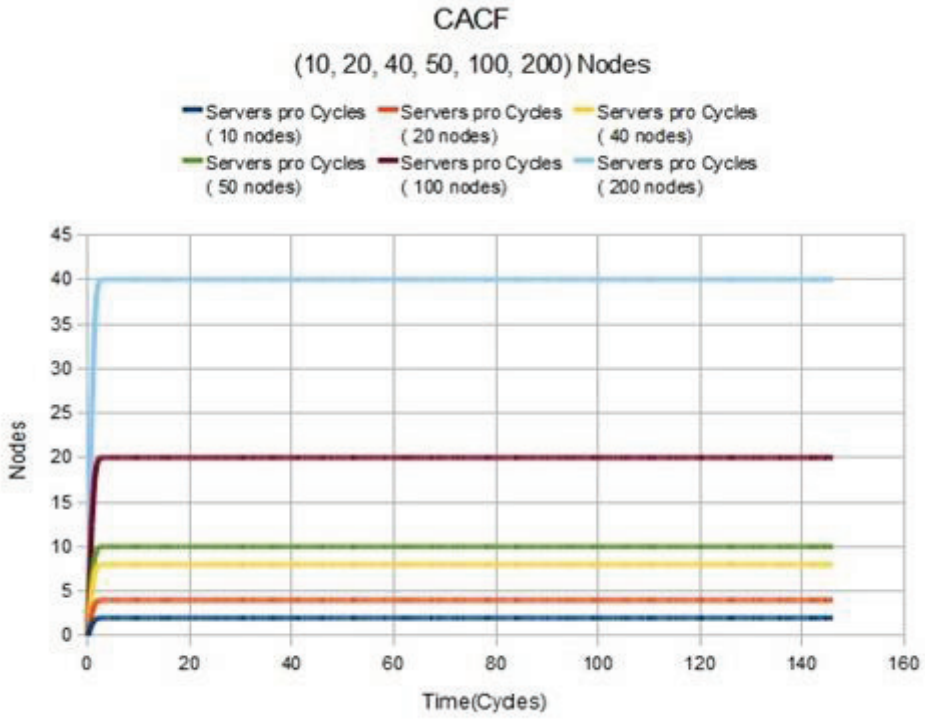

Figure 7-1: Servers availability over the network (CACF)

All curves in figure (7-1) show that initially, there is no server available on the network scenarios. After applying the CACF algorithm, the number of servers will be increased to a specific number after the first cycle and then it will remains constant over time. Figure (7-1) also shows that the number of servers is proportional to the number of nodes that means; when the number of nodes is increased the number of servers will grow in the network.

7.2 SONDe Evaluation (h = 2 hop, h = 4 hop)

To evaluate SONDe performance, we implemented SONDe using the Common-Sim simulator in a cycle based configuration. Cycle can be defined as the time required by all nodes of the network, to achieve their verification process. Following results are based on (40, 50,100,200) nodes in the network. Our simulated scenarios are characterized by two aspects: first, nodes do not move. Second, not all nodes are connected together.

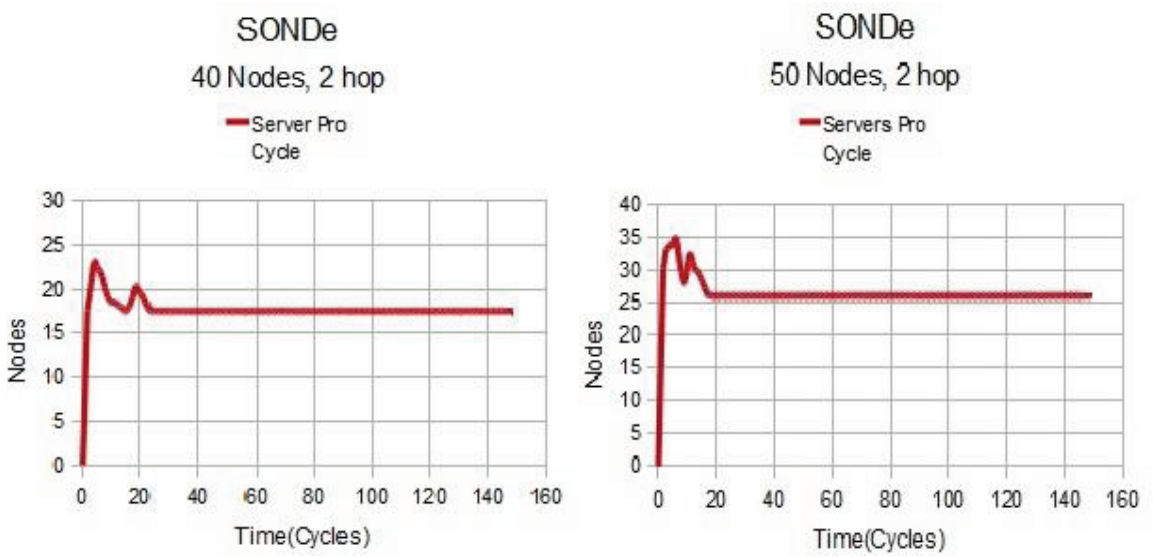

Figure 7-2: Stability graph with (40, 50) nodes, (h = 2 hop)

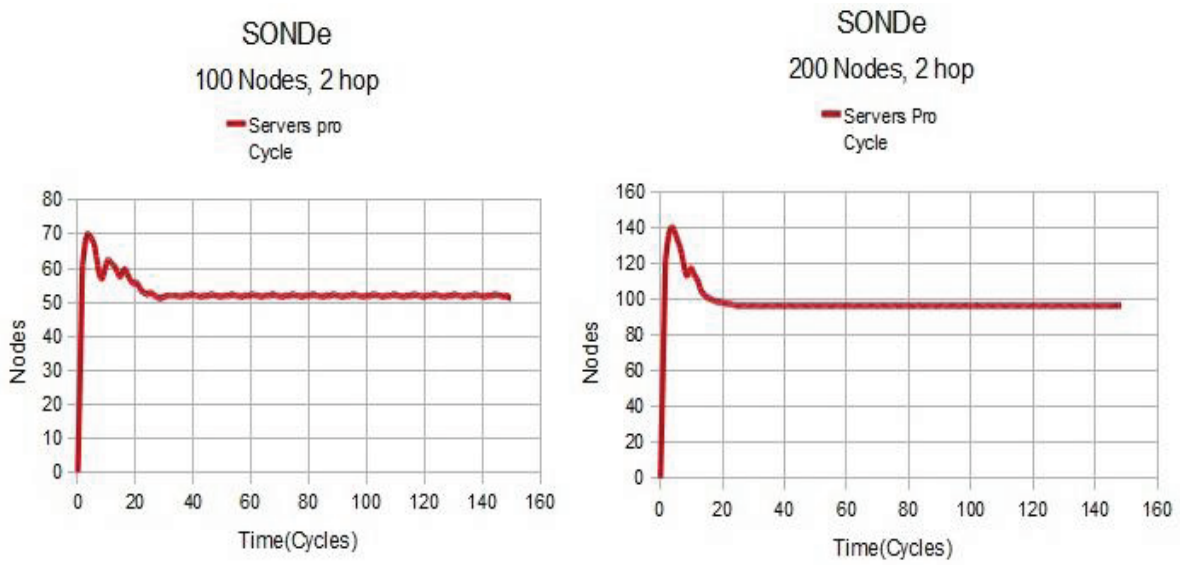

Figure 7-3: Stability graph with (100, 200) nodes, (h = 2 hop)

For better explanation, the results from simulated scenarios are set apart into four groups. The first group shown in Figure (7-2) includes the simulation results of the scenarios (40) and (50) nodes with (h) equal to (2) hops. Figure (7-3) demonstrates the simulation results for (100) and (200) nodes scenarios with (h) equal to (2) hops. Third and fourth groups are exposed in Figure (7-4, 7-5) and contain the evaluation results of the scenarios (40, 50, 100, and 200) nodes with (h) equals to (4) hops.

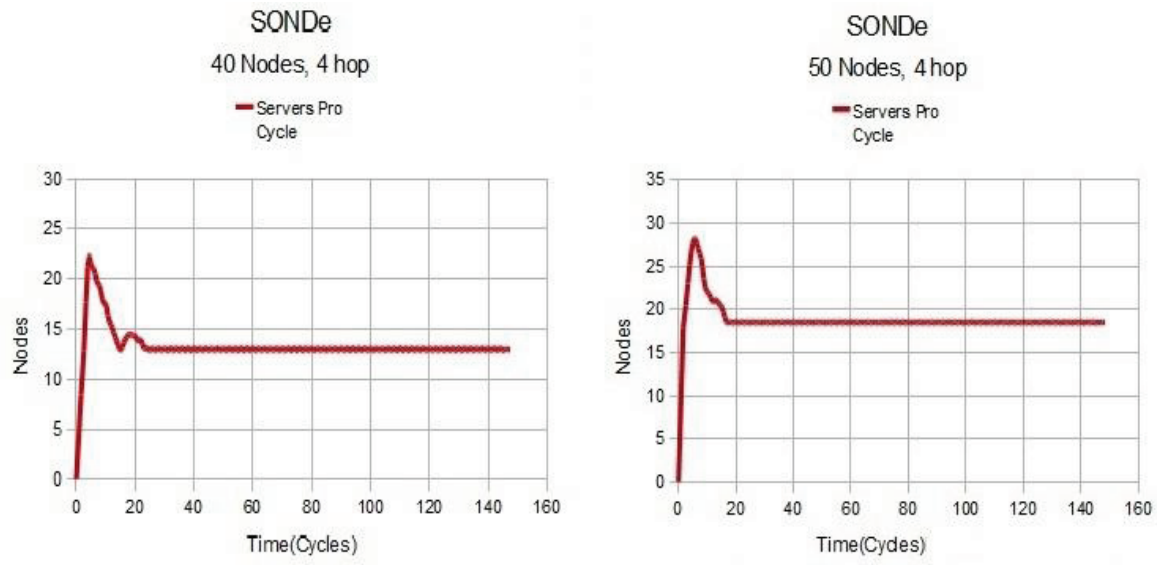

Figure 7-4: Stability graph with (40, 50) nodes, (h = 4 hop)

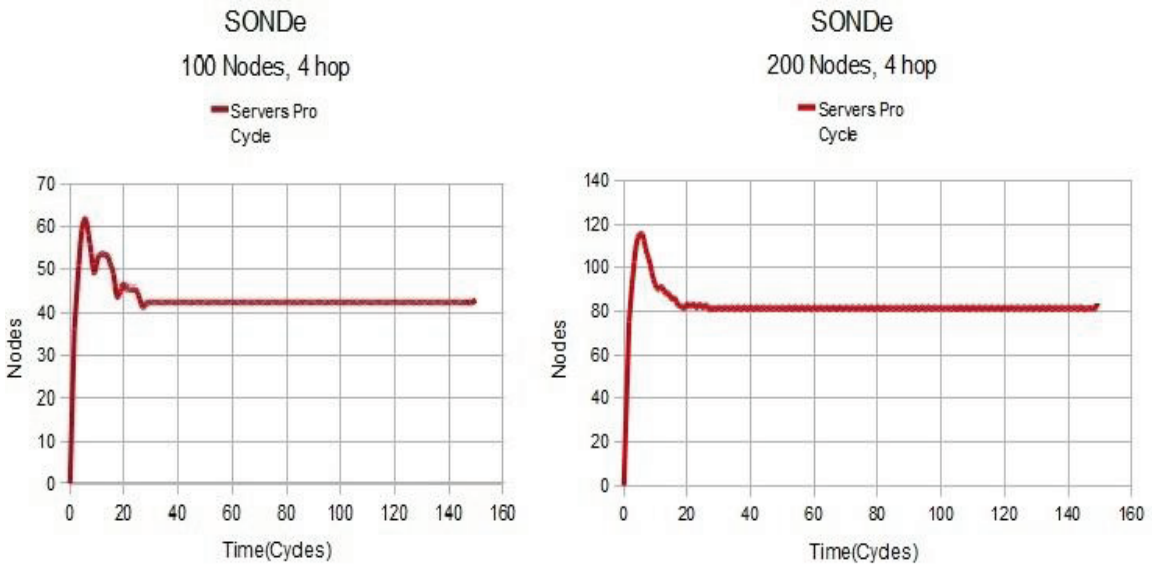

Figure 7-5: Stability graph with (100, 200) nodes, (h = 4 hop)

In Figures (7-2, 7-3, 7-4 and 7-5) you can see that, at the beginning of the SONDe implementation, no providers exist in the network. After that, at each cycle all the nodes will validate simultaneously, if a provider is existed in their h-neighborhood. In the absence of servers some nodes will change their functionalities to act as servers. As a result, a number of servers in the network will increase as shown in the Figures.

After several cycles the number of servers will start to decrease until it reaches a steady-state. As a consequence, it's clear that SONDe algorithm arrives quickly (after a few cycles) at the state of stability, where each node will be served by a nearby server, and the distance between each two servers will be equal to or greater than (h +1).

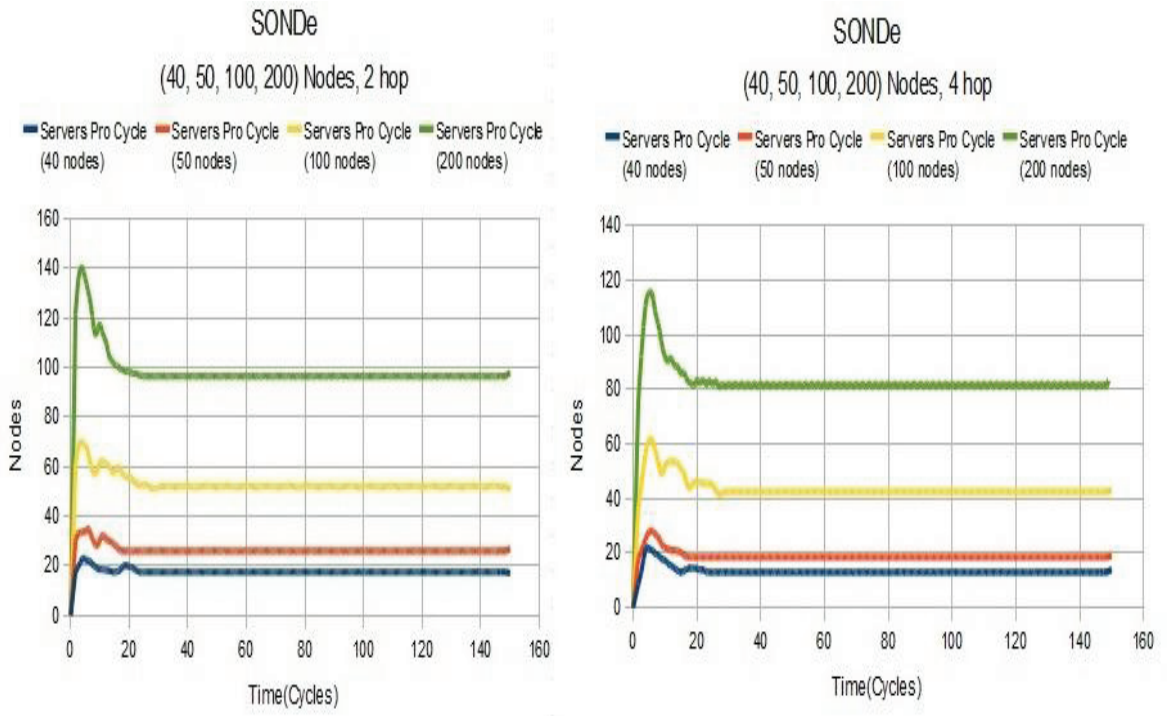

Figure 7-6: SONDe self stabilizes (40, 50,100 and 200) nodes, (h = 2, 4 hop)

All curves in Figure (7-6) display that regardless of high concurrency; only a small number of cycles are needed for the network scenarios to be stabilized. Some small oscillations may appear on the stability curve. We can notice this in the scenario (200 nodes and hop = 4). These oscillations occur due to some nodes' transformation from servers into clients, in case of, (older servers were found in their h-neighborhood – the servers suffer from under load). Or vice versa in case of, servers are suffering from overload situation. In addition to that these oscillations are related to the topology of scenarios.

Figure (7-6) also shows that with a big number of nodes, servers will grow and if (h) gets bigger the number of servers will be decreased.

7.3 SO-RAP Evaluation

Evaluating the performance of SO-RAP algorithm is performed by using CommonSim Simulator. The following number of nodes (30, 40, 50, and 100) is used in the evaluation process. The specified ratio has a value equal to (5%). It can happen that more or less than (5%) of the nodes can become servers.

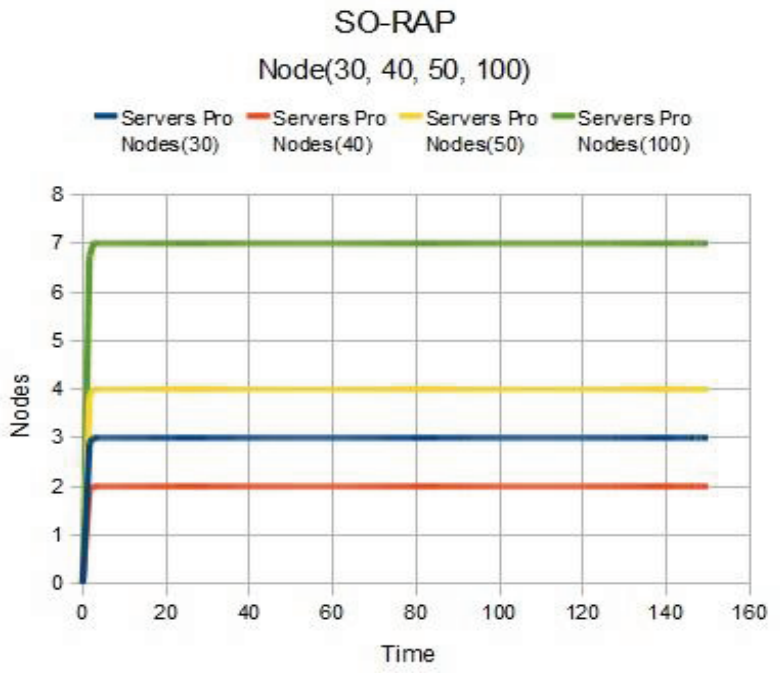

Figure 7-7: Servers availability over the network (SO-RAP)

All curves in figure (7-7) demonstrate that at first, no server was existed in the network, after applying the SO-RAP algorithm and during the first cycle, the number of servers will be increased to a specific number, then it will remains constant over the time.

The availability of servers is not ensured by this algorithm, Because of the random generation of values, thus a possibility that these values are greater than the reference value is existed. The number of servers is changed at every time this algorithm is implemented; it can be more or less than (5%) of the nodes assumed to become servers.

It's obvious in Figure (7-7) that the number of servers is not proportional to the nodes number e.g. the red curve represents (40) nodes scenario, that has only two servers, while the blue curve represents (30) nodes scenario that has three servers. This algorithm is based on chance.

7.4 SO-ARAP Evaluation

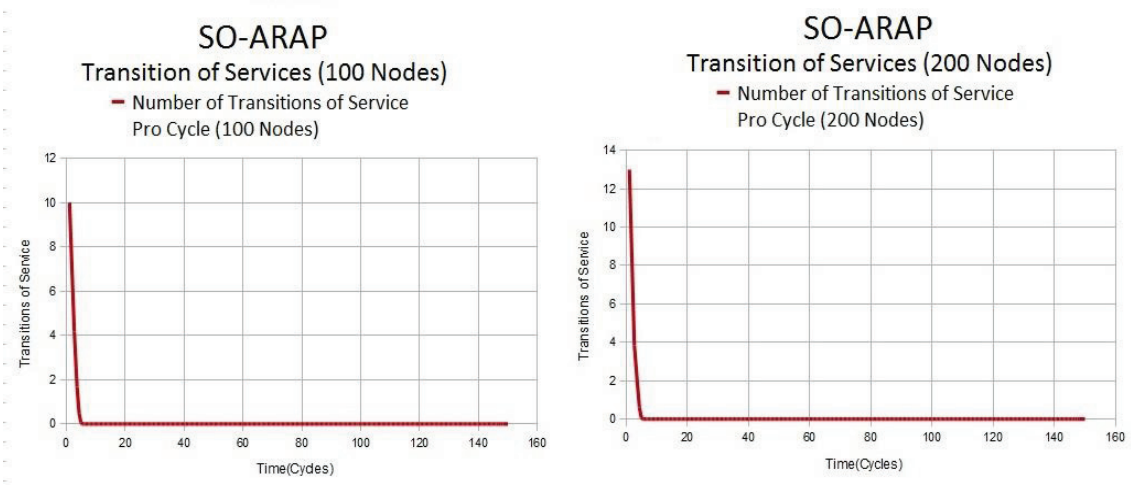

Figure 7-8: The number of service's transitions over time, (100, 200) nodes

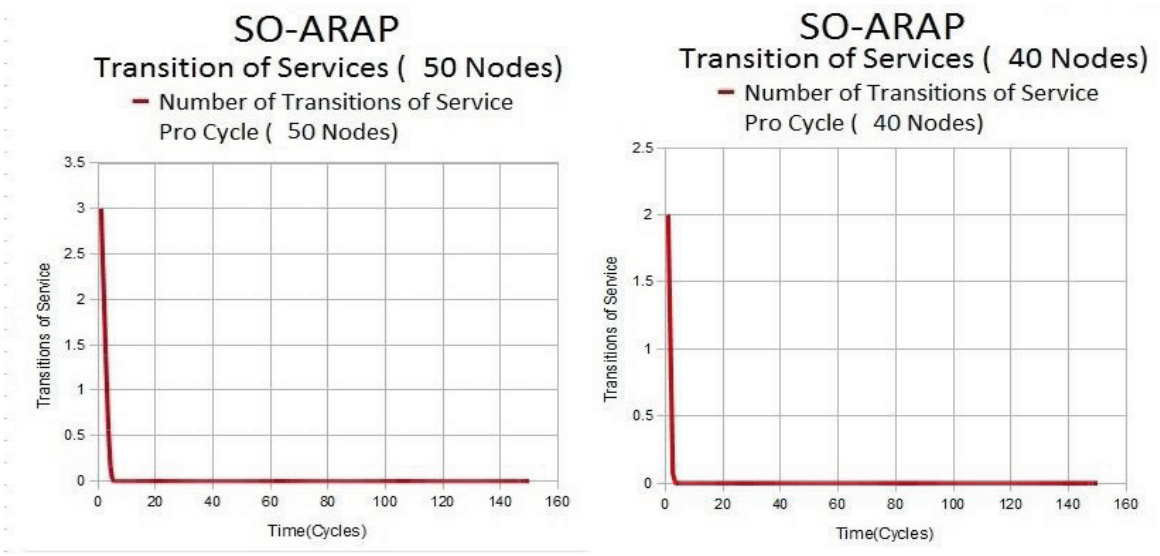

Figure 7-9: The number of service's transitions over time, (40, 50) nodes

Depending on the CommonSim simulator and the following network scenarios (40, 50, 100, and 200 nodes) the evaluation of SO-ARAP is achieved. All curves in figure (7-8) and figure (7-9) represents the number of transitions that the

service instances are taking from the original places where its located in, at first by using SO-RAP algorithm, to another places within its h-neighborhood that are more suitable to host this services. All curves show that the number of these transitions will be at their highest value during the first cycle. Over time, the number of these transitions will be decreased until it reaches a steady-state.

The number of transitions is related to the number of services that are located on the network, and the number of clients that are served by these services.

7.5 Comparison

This section will illustrate a comparison between three service placement algorithms implemented in this thesis. As for SO-RAP it wasn't mentioned because it is implied in SO-ARAP. Then the advantages and disadvantages of each algorithm will be mentioned. Finally, a table that represents the differences between these algorithms will be given.

7.5.1 Hops' average (CACF, SONDe and SO-ARAP)

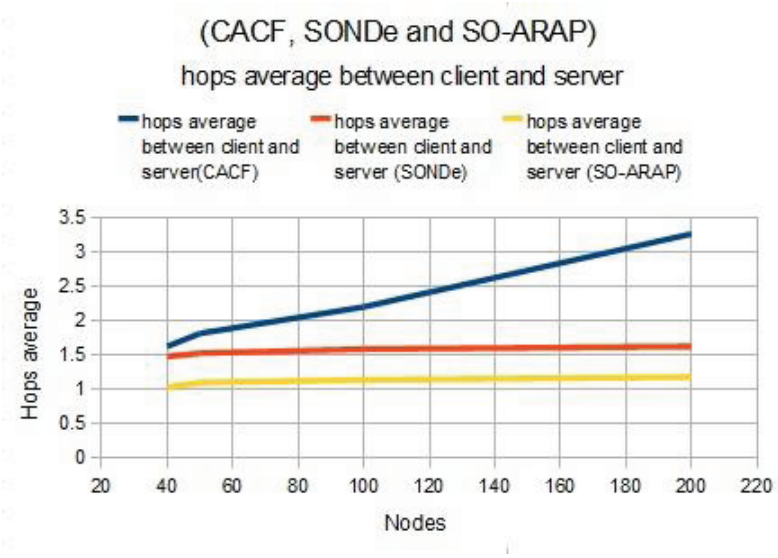

Figure 7-10: Hops' average (CACF, SONDe, and SO-ARAP)

The evaluation process of (CACF, SONDe and SO-ARAP) had been accomplished depending on CommonSim simulator and the following network scenarios (40, 50, 100, and 200 nodes). The hops average between client and server for these algorithms will be demonstrated by all curves in figure (7-10). The blue curve presents the average of hops between the client and the server in CACF algorithm. This curve shows that when the number of the nodes in the network grows up, the hops' average will be increased, this is resulted from the

fact that each client will choose the appropriate server to deal with, depending on the transport cost and regardless of the number of hops in the network e.g. (a client will deal with a server that is three hops away from it instead of a server that is one hop away, because it has the less transport cost). The number of servers that represent about fifth of the total nodes' number, in addition to their distribution on the network will have an effect on the hops average.

As for SONDe Algorithm, the changes of the hops' average between the server and the client will increase slightly when the number of nodes grows up in the network as shown in figure (7-10). this situation is a result of the fact that the relation between the client and the server occurs locally; that means each server will deals with clients that are away from it a distance that equals or less than (h) hop e.g. (h=4, hops average equals to (1.6)). As a result of the increment of h-parameter value, the number of servers will decrease, and then the hops average between the server and client will increase. For example the hops average when (h=2) is less then (h=4). In addition to that, the network topology has an effect on the hops average between server and client.

Regarding SO-ARAP algorithm, which is an improvement of SO-RAP algorithm. The yellow curve will show a sort of stability in the changes of the hops' average, among the servers and the clients by growing up the nodes in the network. This is resulted from the local relation between the clients and the servers; that means every server will deals with clients located in its h-neighborhood. In addition to what mentioned before, servers' movement will help in locating them on the most suitable nodes on the network and this will make them closer to clients.

7.5.2 Clients' average per server (CACF, SONDe and SO-ARAP)

The Clients' average per each server of (CACF, SONDe AND SO-ARAP) algorithms will be illustrated by all curves in figure (7-11). The blue curve presents the average of clients per each server in CACF algorithm for different scenarios. This curve will show that the average of clients served by each server will be stable in all scenarios, because the number of servers is increased in a proportional way with the number of nodes grown up in the network (fifth of the total number of nodes in the network will be servers).

As for SONDe Algorithm, the changes of the clients' average per each server will show a kind if stability when the number of nodes grows up in the network. The number of servers in the network is related to the (h) parameter (distance between clients and servers should be <= h). As this parameter increases, the number of servers will decrease. As a result, the number of clients served by each server will grow up .e.g. figure (7-11) shows two curves with two different h

values (when h=2, the number of clients served per server will equals (2.2) clients, but when h=6, the number of clients served by each server will be equal (3.5)).

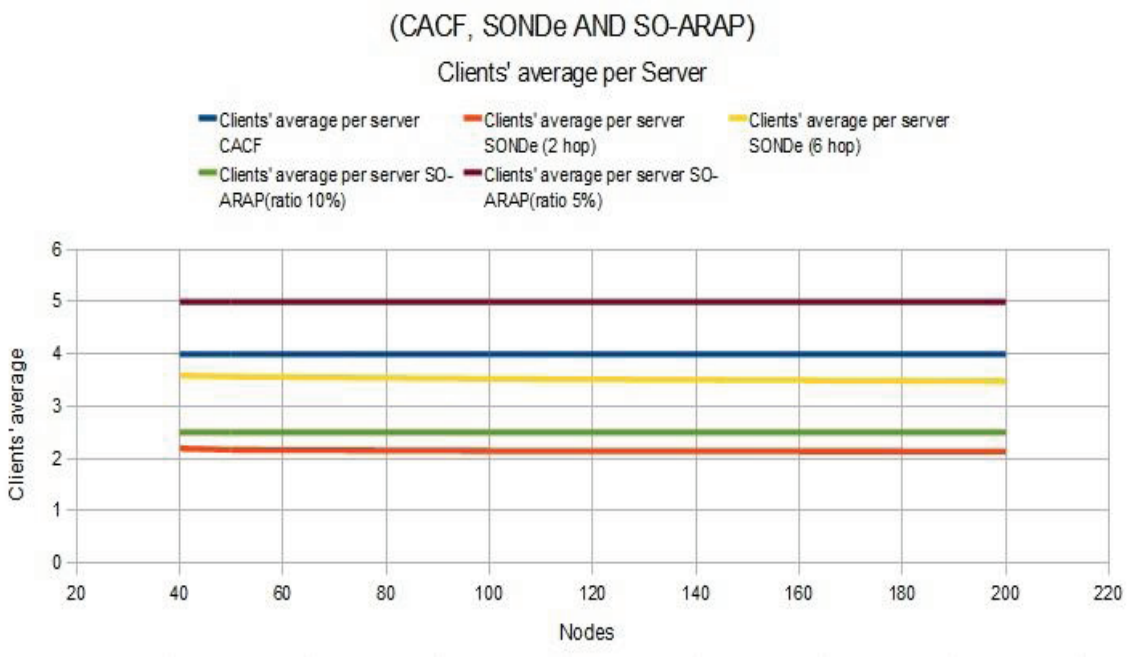

Figure 7-11: Clients' average per Server (CACF, SONDe, and SO-ARAP)

Regarding SO-ARAP algorithm, the curves that represent the average of clients served by each server will remain constant, whenever the nodes' number is increased in the network. The number of servers in the network is based on the ratio value that we choose.

This value is in charge, to determine the number of nodes that will become servers in the network. Figure (7-11) displays two curves with two different ratio values (when ratio = (5%) with (25%) of the nodes are client, the number of clients served per server will equals (5) clients, but when ratio = (10%) with (25%) of the nodes are client, the number of clients served per server will equals (2.5) clients).

Notice: the number of servers in SO-ARAP algorithm may be less or more than the chosen ratio, which will affect the average of clients served by each server. If the number of servers increased over the chosen ratio, the average of clients served by each server will decrease. Vice versa, the average of clients served by each server will increase. As a result of that, the curve that represents the clients' average per server will not be stable as mentioned above, and it will change according to the changes in servers' number.

7.5.3 Algorithms' Advantages and Disadvantages

Centralized service placement Algorithm based on Cost Function (CACF):

- Advantages:
  - Availability of servers (each client will be served).
  - Taking into consideration the resources consumption for each node e.g. CPU cost, memory cost. (Service instances will be placed on the nodes that have the best resources).
  - Taking into consideration the transport cost between clients and servers. (Clients will communicate with servers through links that have the best bandwidth).
- Disadvantages:
  - Communication overhead due to messages exchange between nodes.
  - Not scalable (not suitable for big network).
  - Does not take into consideration under or overload on the servers.

Self-Organizing Network Density (SONDe):

- Advantages:
  - Availability of servers over the network (each client will be served).
  - Taking into consideration under or overload on the servers.
  - Self organized (no central coordination).
  - Stability (the number of servers will be stable after a period of time this will save the resource for the whole network).
  - Scalability.
- Disadvantages:
  - Communication overhead due to messages exchange between nodes.
  - Does not take into consideration the resources consumption of each node.
  - Does not take into consideration the transport cost between clients and servers.

Self-Organized Random Algorithm based on Probability (SO-RAP):

- Advantages:
  - Self organized (no central coordination).
  - Simplicity.
  - No Communication overhead (no messages exchange between nodes).

- Reasonable distribution of the Servers.

- Disadvantages:
  - This algorithm will suffer from a drawback, that in some cases, none of the nodes over the network will act as a server.
  - Does not take into consideration under or overload on the servers.
  - Does not take into consideration the resources consumption of each node.

Self-Organized Advanced Random Algorithm based on Probability (SO-ARAP):

- Advantages:
  - Self organized (no central coordination).
  - Simplicity.
  - Reasonable distribution of the Servers.
  - Takes into consideration the resources consumption of each node.
  - Takes into consideration the transport cost between clients and servers.
- Disadvantages:
  - This algorithm will not ensure that all clients will be served.
  - Does not take into consideration under or overload on the servers.

The following table clarifies the differences between the four algorithms:

	CACF	SONDe	SO-RAP	SO-ARAP
Take into consideration Transport cost between Client and server	✓	✗	✗	✓
Take into consideration The Load on Servers	✗	✓	✗	✗
Self-organized (no central coordination)	✗	✓	✓	✓

Centralized	✓	✗	✗	✗
Use of Verification Process	✗	✓	✗	✓
Scalability	✗	✓	✓	✓
Take into consideration Resource consume (CPU, Memory costs)	✓	✗	✓	✓
Are all the clients Served by servers?	✓	✓	✗	✗

Table 7-1 Comparison between the implemented Algorithms

8 Conclusion & Future Works

8.1 Conclusion

The main goal of this work is the implementation, evaluation and comparison of different service placement algorithm for communication networks. First, a Centralized service placement algorithm based on cost function, which is referred to as "CACF" was developed. The functionality of this algorithm can be described by calculating the cost function of each node in the network.

This cost function represents the availability of resources of this node such as (CPU cost, Memory cost...), and then the most suitable nodes will be chosen to host the service instances. Evaluation process had shown that the number of servers will be fixed, but proportional to the total number of network's nodes. Depending on the transport cost between it and the server, each client selects its appropriate server. The drawbacks of this algorithm are: it doesn't take load on server into consideration; it's not scalable and it has communication over head.

The next step was to implement and evaluate a selected algorithm from the literature. (Self-Organized Network Density) SONDe is a service placement algorithm, where each node of the network periodically verifies the presence or absence of a server in its h-neighborhood. If no server was existed the node will acts as a service provider.

This algorithm ensures the availability of servers, which means every client over the network will be served. The under and overload problem on the servers has been taken into account in SONDe.

SONDe evaluation results had shown that the number of servers will be stable after a period of time. Reducing the number of servers will help to decrease resources consumption of the whole network. The drawbacks of this algorithm are the resources consumption on each node (which is not taken into consideration), and the communication overhead. Depending on the up-mentioned properties we can consider that SONDe algorithm is one of the most important algorithms studied in this thesis.

Probability Random Algorithm (SO-RAP) was implemented and evaluated as the third algorithm in our work. The operation principle of this algorithm can be clarified as follows: each node in the network will compare independently, the value between 0 and 1 that is generated randomly with a reference value, which is equal to all nodes.

The results of this comparison will identify the nodes that will act as servers. This algorithm is characterized by the following specifications such as simplicity, self-organization (no central coordination), no communication between the nodes and reasonable distribution of the servers.

Because of their reliance on random process, this algorithm will suffer from a drawback, when in some cases we will face the fact that; none of the nodes over the network will operate as a server.

SO-ARAP is the fourth algorithm, which is based on a random placement of the servers over the network depending on the previous algorithm (SO-RAP). Then it verifies if the movement of the server from its original place is useful. Each server checks for its neighborhood (within h hops): the costs of server on its current Node, and the costs of server if one of the neighbor's nodes instead of the current one will become a server. If the establishment of the server on a neighbor's nodes is cheaper, the server will be shifted there.

Evaluation results showed that the number of transitions of service between its (h) neighborhood's nodes will be stable after a period of time.

SO-ARAP is characterized by the following proprieties: being self-organized (no central coordination), and the services will be placed on the appropriate nodes (nodes with the most available resources). On the contrast it may face a problem in which, not all the clients will be served.

8.2 Future Work

Some ideas, proposed during the implementation and evaluation of the up-mentioned service placement algorithms in this thesis, deserve further attention. For example, ensuring the availability of service instance over the network in SO-RAP algorithm while, maintaining its self-organization mode of operation. Where this algorithm suffers from a drawback that in some cases, none of the network nodes become a server because of the random value generated by each node is greater than the reference value.

Another example of things that can be improved in the future in this work is, the problem that will face the SO-ARAP algorithm as a result of the random distribution of service instances on the network, in which not all clients will be served. Accordingly, this algorithm should be developed so that service instances are available to all clients in the Network. Furthermore, three of the algorithms developed in this diploma (CACF, SO-RAP and SO-ARAP) should be represented in a mathematical model.

Finally, as we mentioned before the implementation and evaluation of all algorithms was performed depending on the CommonSim simulator, which is developed by the integrated communication system group in TU Ilmenau, This will provide ease and flexibility for any process of development anticipated in the future.

Theses

1. Service placement can be identified as the problem of choosing the most appropriate nodes in the network to be the hosts of the services, which respond to the demands from client's nodes.

2. Locating the services on the optimal places over the network will leads to a decrease in the data traffic, and will enhance the connectivity between clients and servers.

3. The Cost Function represents the availability of resources such as (CPU cost, Memory cost...) of each node on the network.

4. CACF algorithm is a solution presented in this thesis as a way to face the problem of placing services on the network in a centralized way. It depends basically on the cost function to select the most appropriate nodes to host the services. Each client will deal with the server that is accessible through the less transport cost.

5. The average of clients per each server in CACF algorithm will stay constant, because the number of servers is increased in a proportional way with the number of nodes grown up in the network (fifth of the total number of nodes in the network will be servers).

6. In CACF algorithm: while growing up the number of the nodes in the network, the hops' average will be increased, this is resulted from the fact that each client will choose the appropriate server to deal with, depending on the transport cost and regardless of the number of hops in the network.

7. CACF suffers from the following drawbacks: it doesn't take under or overload applied on server into consideration; it's not scalable and it has communication overhead due to the message's exchange between the nodes.

8. SONDe algorithm is a self-organized solution for the service placement problem that ensures the availability of servers; which means that every client over the network will be served. This algorithm also takes into account the under and overload applied on the servers.

9. Stability of servers' number after a period of time; is one of the most properties that characterize the SONDe. It has a good impact on the decrease of the resources consumption of the whole network.

10. The number of servers in SONDe is related to the (h) parameter (distance between clients and servers should be <= h). As this parameter increases, the number of servers will decrease. As a result, the number of clients served by each server will grow up.

11. The hops average in SONDe depends on, the fact that the relation between the client and the server occurs locally, that means each server will deals with clients that are away from it a distance that equals or less than (h) hop, in addition to that the hops average between the client and server change according to the modification that occurs in h-parameter.

12. SONDe drawbacks imply that the availability of resources on each node is not taken into consideration and the communication overhead is existed.

13. SO-RAP is a self-organized solution for placing services over the network that is based on the comparison between a value generated randomly and a fixed reference value on every node.

14. The following properties characterize the SO-RAP algorithm: simplicity, self-organization (no central coordination), no communication between the nodes and reasonable distribution of the servers.

15. The main drawbacks of SO-RAP are as follow: in some cases, none of the nodes over the network will act as a server. Under or overload applied on the servers, as well as the resources consumption of each node are not take into consideration.

16. SO-ARAP is a development of the SO-RAP algorithm achieved through giving the services that are located on the network the possibility to move to other nodes within their h-neighborhood. If the establishment of the service on a neighbor's nodes is cheaper than where it's located, the server will be shifted there. The number of transitions of service between its (h) neighborhood's nodes will be stable after a period of time.

17. The hops average in SO-ARAP algorithm is based on the local relation between the clients and the servers; that means every server will deals

with clients located in its h-neighborhood. In addition to what mentioned before, servers' movement will help in locating them on the most suitable nodes on the network and this will make them closer to clients.

18. In SO-ARAP the average of clients served by each server will remain constant, whenever the nodes' number is increased in the network. The number of servers in the network is based on the ratio value that we choose. Notice: the number of servers in SO-ARAP algorithm may be less or more than the chosen ratio, which will affect the average of clients served by each server.

19. SO-ARAP is identified by the following proprieties: self-organization (no central coordination), services will be placed on the appropriate nodes within their h-neighborhood. On the contrary it may face a problem in which, not all the clients will be served.

Abbreviations

API	Application Programming Interface
POM	Project Object Model
JDK	Java Development Kit
WSN	Wireless Sensor Networks
SOAs	Service Oriented Architectures
UFLP	Uncapacitated Facility Location Problem
CACF	Centralized Algorithm based on Cost Function
SONDe	Self-Organizing Network Density
SO-RAP	Self-Organizing Random Algorithm based on Probability
SO-ARAP	Self-Organized Advanced Random Algorithm based on Probability

List of Figure

List of Tables

Bibliography

[1] C. Matthew Mackenzie, Ken Laskey, Francis McCabe, Peter F. Brown, Rebekah Metz, and Booz Allen Hamilton (Eds.). Reference Model for Service Oriented Architecture OASIS Standard, October 2006.

[2] A Survey (study) of Current Directions in Service Placement in Mobile Ad-hoc Networks, Georg Wittenburg, Jochen Schiller, Department of Mathematics and Computer Science, freie Universität of Berlin, Takustr.9, 14195 Berlin, Germany.

[3] Georg Wittenburg, Service Placement in Ad Hoc Networks, Doctoral Dissertation, freie Universität Berlin, 27. September 2010.

[4] A Survey of Services Placement Mechanisms for Future Mobile Communication Networks, Shahzad Ali, Andreas Mitschele-Thiel, Ilmenau University of Technology.

[5] Self-Organization in Mobile Communication Systems, Andreas Mitschele Thiel, Ilmenau University of Technology.

[6] Wireless World Research Forum (WWRF), official webpage: www.wireless- world-research.org, accessed at November, 04 2009.

[7] Self-Organization Methodologies for Services Placement in Future Mobile Communication Networks, Shahzad Ali, Andreas Mitschele-Thiel, Ilmenau University of Technology.

[8] Pitu B. Mirchandani and Richard L. Francis, editors. Discrete Location Theory. Wiley-Interscience, December 1990.

[9] S. L. Hakimi. Optimum Distribution of Switching Centers in a Communication Network and Some Related Graph Theoretic Problems. Operations Research, 13(3):462–475, May 1965.

[10] http://en.wikipedia.org/wiki/Linear_programming.

[11] G. Corneújols, G. L.Nemhauser, and L. A. Wolsey. The Uncapacitated Facility Location Problem. In Pitu B. Mirchandani and Richard L. Francis, editors, Discrete Location Theory, chapter 3. Wiley-Interscience, December 1990.

[12] Josh Reese. Solution Methods for the p-Median Problem: An Annotated Bibliography. Networks, 48(3):125–142, August 2006.

[13] Artur Andrzejak, Sven Graupner,Vadim Kotov, Holger Trinks, Algorithms for Self-Organization and Adaptive Service Placement in Dynamic Distributed Systems, Internet Systems and Storage Laboratory, HP Laboratories Palo Alto, September 17th, 2002.

[14] Vincent Gramoli, Anne-Marie Kermarrec, Erwan Le Merrer, SONDe, a Self-Organizing Object Deployment Algorithm in Large-Scale Dynamic Systems, Seventh European Dependable Computing Conference, 2008.

[15] https://ih55.theoinf.tu-ilmenau.de/sites/CommonSim/installation.html

Appendix

A Source structure of SONDe

The StartSimulation_1.java file:

```
package test;
import de.tuilmenau.ics.CommonSim.core.Environment;
import de.tuilmenau.ics.CommonSim.core.IEvent;
import java.util.Random;
import java.util.Arrays;
import java.util.Scanner;
import java.io.Console;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.io.File;
import java.util.*;

public class StartSimulation_1 implements IEvent
{
  private int n;
  private String Name;
  private int currenthop ;
  private int Hop;
  private String Server;
  private String Client;
  private boolean Find_Server;
  private boolean FindServer;
  private boolean ServerisFound;
  private int ServerName = 0;

  int [][]  ConnArray;
  String   Charray[];
  int       HLocal[];
  long     ServerOld[];
  int       over_under_load[];
  int       over_under_load_Cycles = 0;
  int       Repeat = 0;
  int       CyclesArray[];
  int       ThresholdArray[];
  int       UnderLoadCycle[];
  int       OverLoadCycle[];
  int       Nodes[];
  int       currenthopArray[];
  int       ServerNumbers[];
  boolean  registerd[];

  @Override
  public void fire()
   {
```

```
System.out.println("Event got scheduled at:
"+Environment.get().getKernel().getCurrentTime().toString());
Scanner in = new Scanner(System.in);
Console console = System.console();
Random  generator = new Random();

System.out.println("");
System.out.println("The number of  Nodes, that used as a default in our Scenarios are: 5, 10, 20, 30,
40, 50, 100, 200");
System.out.print("Enter the Chosen number: ");
n = in.nextInt();

Charray = new String[n];
HLocal = new int[n];
ServerOld = new long[n];
over_under_load = new int[n];
Node_1 node = new Node_1();
registerd = new boolean[n];
UnderLoadCycle = new int[n];
OverLoadCycle = new int[n];
Nodes = new int[n];
currenthopArray = new int[n];
ServerNumbers  = new int[200];

 while ((n != 5)&&(n != 10)&&(n != 20)&&(n != 30)&&(n != 40)&&(n != 50)&&(n != 100)&&(n !=
200))
{
 System.out.print ("Enter again the number of nodes from the given list :");
 n = in.nextInt();
}
switch(n)
        {
        case 5:{ ConnArray = new int [][]
                {
                  {0 , generator.nextInt(100)+1, 0, 0, 0},
                  {generator.nextInt(100)+1, 0, generator.nextInt(100)+1, 0, 0},
                  {0, generator.nextInt(100)+1, 0, generator.nextInt(100)+1, 0},
                  {0, 0, generator.nextInt(100)+1, 0, generator.nextInt(100)+1},
                  {0, 0, 0, generator.nextInt(100)+1, 0}
                 };

                 };
              break;
        }
System.out.println();
System.out.print("Enter a String or Character that will be used as a name for Nodes: ");
Name = console.readLine();
System.out.println("");
System.out.print("Result is stored in OutStartSimulation_1.txt file");
System.out.println("");
System.out.println("");
try
    {
      File file = new File("OutStartSimulation_1.txt");
      FileOutputStream fos = new  FileOutputStream(file);
```

```
            PrintStream ps =new PrintStream(fos);
            System.setOut(ps);

            System.setErr(ps);
            throw new Exception("text Exception");
          }
catch(FileNotFoundException fnfEx)
          {fnfEx.printStackTrace();}

catch(Exception Ex)
          {Ex.printStackTrace();}
for(int i=0; i<n;i++)
         { String Inf = node.BecomeClient();
          Charray[i]  = Inf;
          over_under_load[i] = 0;
          Nodes[i] = i;
         }
shuffleArray(Nodes);
for(int i : Nodes)
        { for(int l=0; l<n;l++)

           { currenthopArray[l] = 0; }
           ServerisFound = false;
           Hop = 4;

           if(Charray[i] != Server)
           {
            System.out.println(""+Name+i +" is a Client");
            System.out.println("Search for a Server in rang of " +Hop +" Hops");
            System.out.println("If no Server found, this Node will become a Server.");

            for(int j=0; j<n;j++)
              {
               currenthopArray[i] = 0;
               if((ConnArray[i][j] != 0)&&(currenthopArray[i]<Hop)&&(ServerisFound==false))
                    {
                         currenthopArray[j] = currenthopArray[j] + 1;
                         currenthop = currenthopArray[j];
                         if(Charray[j].compareTo("Server")!= 0)
                           {
                                 System.out.println(""+Name+j +" is a "+Charray[j]);
                                 boolean X = SearchForServer(i,j,currenthop,Hop);
                                 if(X==true)  {
                                if(registerd[i] == false) {
                                over_under_load[getServer()] = over_under_load[getServer()] + 1;
                                registerd[i] = true; }

                                 System.out.println("The Load on the Server "+Name+getServer() +"
                                 is:"+over_under_load[getServer()]);
                                 System.out.println(""+Name+i +" is connected to Server in a range of " +Hop
                                 +" Hops ");
                                 System.out.println("");
                                 FindServer = false;
                                 ServerisFound = true;}}
```

```
                                else
                                 {
                                    System.out.println(""+Name+j +" is a " +Charray[j]);
                                   System.out.println(""+Name+i +" is connected to a Server in a range of "
                                   +Hop +" Hops ");
                                    if(registerd[i] == false) {
                                    over_under_load[j] = over_under_load[j] +1;
                                    registerd[i] = true; }
                                    System.out.println("The Load on the Server "+Name+j +"
                                   is:"+over_under_load[j]);
                                    System.out.println("");

                                    ServerisFound = true;
                                  }
                            }
                        }

                  if(ServerisFound==false) {
                    Charray[i]=node.BecomeServer();
                    long time = System.nanoTime();
                    ServerOld[i] = time;
                    System.out.println(""+Name+i +" become a " +Charray[i]+" at time:"+time);
                    System.out.println("");
                    ServerisFound = true; }
      }
    }
ThresholdArray = new int[n];
int Cycle = 0;
for(int i=0; i<n;i++)
   {HLocal[i] = generator.nextInt(5);
    ThresholdArray[i]= generator.nextInt(2)+1;
    over_under_load[i] = 0;
    registerd[i]=false;}

do
 { System.out.println("Step"+Repeat);
   for(int i=0; i<n;i++)
     { ServerisFound = false;
       System.out.println(" ");
       System.out.println(""+Name+i +" is a "+Charray[i]+" and its Hlocal is = "+HLocal[i]);
       if (Charray[i].compareTo("Server")!= 0)
          {
           for(int j=0;j<n;j++)
             {
              currenthop=0;
              Hop = HLocal[i];
              if((ConnArray[i][j] != 0)&&(currenthop < Hop )&&(ServerisFound==false)) {
              currenthop = currenthop + 1;
              if(Charray[j].compareTo("Server")!= 0){
              boolean X = SearchForServer(i,j,currenthop,HLocal[i]);
              if(X==true) {
              if(registerd[i] == false) {
              over_under_load[getServer()] = over_under_load[getServer()] + 1;
              registerd[i] = true;   }
              System.out.println(""+Name+i +" is connected to Server in a range of its h local ");
```

```
            System.out.println("");
            ServerisFound = true; } }
            else {
            if(registerd[i] == false)  {
            over_under_load[j] = (over_under_load[j] + 1);
            registerd[i] = true;
           }
       System.out.println(""+Name+j +" is a " +Charray[j]);
       System.out.println(""+Name+i +" is connected to a Server in a range of its H local ");
       System.out.println("The Load on the Server "+Name+j +" is:"+over_under_load[j]);
       System.out.println("");
       ServerisFound = true;
      }
    }
  }
if(ServerisFound==false)
  { Charray[i]=node.BecomeServer();
   long time = System.nanoTime();
   ServerOld[i]=time;
   System.out.println(""+Name+i +" become a " +Charray[i]+" at time:"+time);
   System.out.println("");
   ServerisFound = true;
  }
else {
        Cycle = Cycle +1;
        if(Cycle >ThresholdArray[i]) {
        HLocal[i] = getNbrAvg(i);
        System.out.println("The new value of Hlocal for "+Name+i +" is:"+HLocal[i]);  }
       }

   }
   else {
   Hop = HLocal[i];
   for(int j=0;j<n;j++) {
   if(Charray[i].compareTo("Server")==0) {
   currenthop=0;
   if((ConnArray[i][j] != 0)&&(currenthop <= Hop )){
   currenthop = currenthop + 1;
   if(Charray[j].compareTo("Server")!= 0){
   boolean D = SearchForOldServer(i,j,currenthop,HLocal[i]);
   if(D==true)
   { System.out.println(""); }
    else{

     if((over_under_load[i]>3)&&(HLocal[i]>0)){
     System.out.println("Server "+Name+i +" is OverLoaded ");
     OverLoadCycle[i] = OverLoadCycle[i] + 1;
     System.out.println("OverLoad Cycle: "+OverLoadCycle[i]);
     UnderLoadCycle[i] = 0; }
     else{
     if((over_under_load[i]==0)&&(HLocal[i]<2)) {
     System.out.println("Server "+Name+i +" is UnderLoaded ");
     UnderLoadCycle[i] = UnderLoadCycle[i] +1;
     System.out.println("UnderLoad Cycle: "+UnderLoadCycle[i]);
     OverLoadCycle[i] = 0;}
```

```
        else{
        UnderLoadCycle[i] = 0;
        OverLoadCycle[i] = 0; }
        }
        if(OverLoadCycle[i] > 3 ) {
        OverLoad(i);
        OverLoadCycle[i] = 0; }
        else if(UnderLoadCycle[i] > 3 ) {
        UnderLoad(i);
        UnderLoadCycle[i] = 0; }
        } }
        else
        {
         System.out.println(""+Name+j +" is a "+Charray[j]);
         System.out.println("There is a Server in the range of H local");
         if((ServerOld[i] > ServerOld[j])){
         System.out.println("This Server is an Older Server in the range of Hlocal");
         if((HLocal[j]==HLocal[i])){
         System.out.println("The Hlocal values for theses two Servers are equal");
         String Inf = node.BecomeClient();
         Charray[i]  = Inf;
         System.out.println(""+Name+i +" become a Client");
         }
         else
         {
          System.out.println("The Hlocal values of these two Servers are not equal");
         }
         }
         else
         {System.out.println("This Server is not an Older Server in the range of Hlocal");
         }
    }
    }
    }
    }
    }
    }
System.out.println(" ");
int Servernumber = 0;
for(int y = 0; y<n; y++)
  {
    if (Charray[y].compareTo("Server") == 0)
    {Servernumber = Servernumber + 1;}
  }
System.out.println("Servernumber: "+ Servernumber);

ServerNumbers[Repeat] = Servernumber ;
System.out.println("Repeat: "+Repeat);
Repeat = Repeat + 1; }
while(Repeat<200);
for(int y = 0; y<200; y++){
System.out.println(""+ServerNumbers[y]); }
}
```

```
public boolean SearchForServer(int x,int y,int z,int t)
    {
     int currenthop = z;
     for(int s=0;s<n;s++){
     if ((ConnArray[y][s] != 0) && (s!=x)&& (currenthop<t)&&(FindServer != true))
       {
        if (Charray[s].compareTo("Server")!=0)
          {
            System.out.println("" +Name+s +" is a " + Charray[s]);
            currenthopArray[s] = currenthop+1;
            int currenthop_a = currenthopArray[s];
            SearchForServer(y,s,currenthop_a,t);
          }
          else
          {
            System.out.println(""+Name+s +" is a "+Charray[s]);
            SetServer(s);
            FindServer = true;
          }
       }
     }
     Find_Server = FindServer ;
     return  Find_Server;
    }

public boolean SearchForOldServer(int x,int y,int z,int t)
    {
     boolean FindOldServer = false;
     Node_1 node = new Node_1();
     int currenthop = z;
     for(int s=0;s<n;s++){
     if ((ConnArray[y][s] != 0) && (s!=x)&& (currenthop<t)) {
     currenthop=currenthop+1;
     if (Charray[s].compareTo("Server")!=0)
       {
        SearchForOldServer(y,s,currenthop,t);
       }
     else{
     System.out.println(""+Name+s +" is a "+Charray[s]);
     System.out.println("There is a Server in the range of H local");
     if((ServerOld[x] < ServerOld[s])){
     System.out.println("This Server is an Older Server in the range of Hlocal");
     if((HLocal[s]==HLocal[x]))
      {
       System.out.println("The Hlocal values for theses two Servers are equal");
       FindOldServer = true;
       String Inf = node.BecomeClient();
       Charray[x]  = Inf;
       System.out.println(""+Name+x +"become a Client");
      }
      else
      {
       System.out.println("The Hlocal values of theses two Servers are not equal");
      }
      }
```

```
        else {
        System.out.println("This Server is not an Older Server in the range of Hlocal");}
       }
       }
       }
        return FindOldServer;
       }

public int getNbrAvg(int r)
  {
    int Avg = 0;
    int num = 0;
    for(int j=0;j<n;j++){
    if(ConnArray[r][j] != 0){
    num = num + 1;
    Avg = Avg + HLocal[j]; }
    return  Avg/num ;
   }

public void OverLoad(int i)
  {
    HLocal[i]=HLocal[i]-1;
    System.out.println(""+Name+i+" new Hlocal is: "+HLocal[i]);
    for(int s=0;s<n;s++){
    if ((ConnArray[i][s] != 0)){
    HLocal[s] = HLocal[i];
    System.out.println(""+Name+s+" new Hlocal is: "+HLocal[s]);}}
   }

public void UnderLoad(int i)
  {
     HLocal[i]=HLocal[i]+1;
     System.out.println(""+Name+i+" new Hlocal is: "+HLocal[i]);
     for(int s=0;s<n;s++){
     if ((ConnArray[i][s] != 0)) {
     HLocal[s] = HLocal[i];
     System.out.println(""+Name+s+" new Hlocal is: "+HLocal[s]);}}
   }

public void shuffleArray(int[] a)
  {
      int length = a.length;
      Random random = new Random();
      random.nextInt();
      for(int i=0;i<n;i++){
      int change = i + random.nextInt(n-i);
      Swap(a, i, change);}
   }
public void Swap(int[] a, int i, int change)
  {
     int x = a[i];
     a[i] = a[change];
     a[change] = x;
   }
public void SetServer(int m)
```

```
    {
     ServerName = m;
    }
  public int getServer()
    {
      return ServerName;
    }

}
```

The Node_1.java File:

```
package test;
public class Node_1
  {
   private String Name;
   private String Cl;
   private String Ser;

   public Node_1()
    {
    }

   public String BecomeClient()
    {
      Cl="Client";
      return Cl;
    }
   public String BecomeServer()
    {
      Ser="Server";
      return Ser;
    }
}
```

A Source structure of SO-RAP

The StartSimulation_2.java file:

```
public class StartSimulation_2 implements IEvent
  {
   private int n;
   private String Name;

   String Charray[];
   int    Nodes[];
   int [][] ConnArray;

   @Override
   public void fire()
     {
       System.out.println("Event got scheduled at:
       "+Environment.get().getKernel().getCurrentTime().toString());
       Scanner in = new Scanner(System.in);
       Console console = System.console();
       Random generator = new Random();
```

```
System.out.println("");
System.out.println("The number of Nodes, that used as a defualt in our Scenarios are: 30, 40, 50,
100");
System.out.print("Enter the Chosen number: ");
n = in.nextInt();
Node_2 node = new Node_2();

Charray = new String [n];
Nodes = new int[n];

while((n != 30)&&(n != 40)&&(n != 50)&&(n != 100))
 {System.out.print("Enter again the number of nodes from the given list: ");
   n = in.nextInt(); }
System.out.println("");
System.out.print("Enter a String or Character that will be used as a name for Nodes: ");
Name = console.readLine();
System.out.println("");

System.out.print("Result is stored in OutStartSimulation_2.txt file");
System.out.println("");
System.out.println("");
try
 {
  File file = new File("OutStartSimulation_2.txt");
  FileOutputStream fos = new FileOutputStream(file);
  PrintStream ps = new PrintStream(fos);
  System.setOut(ps);
  System.setErr(ps);
  throw new Exception("text Exception");
 }
 catch(FileNotFoundException fnfEx)
  {fnfEx.printStackTrace();}

 catch(Exception Ex)
  {Ex.printStackTrace();}
 finally
  {//System.setOut(Console); }

for(int i=0; i<n;i++)
   { String Inf = node.BecomeClient();
     Charray[i]  = Inf;
     Nodes[i] = i;}
shuffleArray(Nodes);
for(int i : Nodes)
 {
  double value =  node.Randomvalue();
  if(value <= 0.05)
    { Charray[i] =  node.BecomeServer();
      System.out.println(""+Name+i +" is a "+Charray[i]);}
  else
    { System.out.println(""+Name+i +" is a "+Charray[i]); } }
 }

public void shuffleArray(int[] a)
```

```
        {
         int length = a.length;
         Random random = new Random();
         random.nextInt();
         for(int i=0;i<n;i++)
            { int change = i + random.nextInt(n-i);
              Swap(a, i, change)}
        }

 public void Swap(int[] a, int i, int change)
      { int x = a[i];
        a[i] = a[change];
        a[change] = x;
      }
 }
```

The Node_2.java File:

```
package test;
import java.util.Random;

public class Node_2
{
 private String Name;
 private String Cl;
 private String Ser;

 public Node_2()
  {}
 public double Randomvalue()
   {
    Random generator2 = new Random();
    double r = generator2.nextDouble();
    return r;
    }

 public String BecomeClient()
    {
     Cl="Client";
     return Cl;
    }
 public String BecomeServer()
    {
     Ser="Server";
     return Ser;
    }
}
```

A Source structure of SO-ARAP

The StartSimulation_3.java file:

```
package test;

import de.tuilmenau.ics.CommonSim.core.Environment;
import de.tuilmenau.ics.CommonSim.core.IEvent;

import java.util.Random;
import java.util.Scanner;
import java.util.Arrays;
import java.util.ArrayList;
import java.io.Console;
import java.util.Collections;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.io.File;
import java.util.*;

public class StartSimulation_3 implements IEvent
  {
   private int n;
   private String Name;
   private int    Hop;
   private String Server;
   private int    ServerCost=0;
   private int    ClientCost=0;

   private boolean FindClient;
   private boolean FindServer;

   String Charray[];
   int    Index[];
   int    Nodes[];
   int    [][] ConnArray;
   int    CpuCost[];
   int    MemCost[];
   int    HopArray[];
   int    HopArray_a[];
   int    HopArray_b[];
   int    CompareArray[];
   int    ServerIndex[];
   int    ClientIndex[];

   String Clients[];
   String NameArray[];
   ArrayList<Integer> list = new ArrayList<Integer>();
   ArrayList<Integer> list1 = new ArrayList<Integer>();


   @Override
   public void fire()
```

```
{
 System.out.println("Event got scheduled at:
 "+Environment.get().getKernel().getCurrentTime().toString());
Scanner in = new Scanner(System.in);
Console console = System.console();
Random generator = new Random();

System.out.println("");
        System.out.println("The number of Nodes, that used as a defualt in our Scenarios are: 30, 40,
 50,100");
System.out.print("Enter the Chosen number: ");
n = in.nextInt();
Node_3 node = new Node_3();
Random generator_Value = new Random();

Index   = new int[1];
Charray = new String [n];
Nodes = new int[n];
CpuCost = new int[n];
MemCost = new int[n];
Clients = new String[n];
HopArray = new int[n];
HopArray_a = new int[n];
HopArray_b = new int[n];
NameArray = new String[n];
ServerIndex = new int[1];
ClientIndex = new int[1];
CompareArray = new int[n];

while((n != 30)&&(n != 40)&&(n != 50)&&(n != 100))
 {
 System.out.print("Enter again the number of nodes from the given list: ");
 n = in.nextInt();

 }
  System.out.println("");
  System.out.print("Enter a String or Character that will be used as a name for Nodes: ");
  Name = console.readLine();

  System.out.println("");
  System.out.print("Result is stored in OutStartSimulation_3.txt file");

  System.out.println("");
  System.out.println("");
  try
  {
   File file = new File("OutStartSimulation_3.txt");
   FileOutputStream fos = new FileOutputStream(file);
   PrintStream ps = new PrintStream(fos);
   System.setOut(ps);
   System.setErr(ps);
   throw new Exception("text Exception");
  }
  catch(FileNotFoundException fnfEx)
    {fnfEx.printStackTrace();}
```

```
   catch(Exception Ex)
     { Ex.printStackTrace();}
   finally
     {  //System.setOut(Console);}

for(int i=0; i<n;i++)
  {
      String Inf = node.BecomeClient();
       if((i%2)==0 && i!=0)
       { Charray[i] = Inf;}
        else{
        Charray[i]="NonClient";}
       NameArray[i] = (""+Name+i);
       CpuCost[i] = generator_Value.nextInt(100)+1;
       MemCost[i] = generator_Value.nextInt(100)+1;
       Nodes[i] = i;
   }
shuffleArray(Nodes);
for(int i : Nodes)
  {
    double value =  node.Randomvalue();
    if(value <= 0.05){
    Charray[i] =  node.BecomeServer();
    System.out.println(""+Name+i +" is a "+Charray[i]);}
    else
     {System.out.println(""+Name+i +" is a "+Charray[i]);  }
   }
for(int i=0;i<n;i++)
 {
       Hop = 2;
       //Search for Clients of specific Server
       if(Charray[i].compareTo("Server")==0)
         {
         ServerIndex[0]=i;
         System.out.println("");
         System.out.println(""+Name+i +" is a "+Charray[i]);

         for(int j=0;j<n;j++)
           {
           int HopCounter = 0;
           if(ConnArray[i][j]!=0 && Charray[j].equals(Server) == false && HopCounter < Hop)
            { HopCounter=HopCounter+1;
              if(Charray[j].compareTo("Client")==0)
               {
                Clients[j]=(""+Name+j);
                Searchfor(i,j,HopCounter,Hop);
               }
              else
              {Searchfor(i,j,HopCounter,Hop); }
              }
              }

              //calculat the Server transportcost
               for(int d =0;d<n;d++)
                {
```

```
FindServer=false;
if(Clients[d] != null)
 {
  System.out.print("");
  System.out.println("This Node is Client:"+NameArray[d]);
  int HopCounter_b = 0;
  for(int k=0;k<n;k++)
   {
    if(ConnArray[d][k] != 0 && FindServer==false)
     {
      String Str1=NameArray[ServerIndex[0]];
      //System.out.println(""+Str1);
      String Str2=NameArray[k];
      //System.out.println(""+Str2);

       if(Str1.equals(Str2))
        {
         int Value1 =0;
         Value1 = ConnArray[d][k];
          //System.out.println(""+Value1);
          FindServer=true;
          list.add(Value1);
         }
         else
          {
           int Value2 = 0;
           Value2 = ConnArray[d][k];
           //System.out.println(""+Value2);
           HopCounter_b = HopCounter_b +1;
           SearchForSer(d,k,Value2,HopCounter_b,Hop);
          }

          }
          }
          }
          }

Object servercost[] = list.toArray();
list.clear();
for(int row =0;row<servercost.length;row++)
  {
   //System.out.print(servercost[row]);
   ServerCost += ((Integer) servercost[row]).intValue();
   System.out.println("");

  }
   System.out.println("the Transport Cost for Server:"+Name+i +" is "+ServerCost);
   System.out.println("The Cpu Cost: "+CpuCost[ServerIndex[0]]);
   System.out.println("The Mem Cost: "+MemCost[ServerIndex[0]]);
   ServerCost = ServerCost + CpuCost[ServerIndex[0]]+ MemCost[ServerIndex[0]];
   System.out.println("the new Transport Cost for Server:"+Name+ServerIndex[0] +" is
   "+ServerCost);


for(int p =0;p<n;p++)
```

```
{
 if(Clients[p] != null)
  {
  ClientIndex[0] = p;
  //System.out.print("");
  //System.out.println(""+p);
  //System.out.println("this node is client:"+NameArray[p]);

  for(int u=0;u<n;u++)
   {
   if(Clients[u] != null && Clients[p] != Clients[u])
    {
    //System.out.print("");
    //System.out.println(""+u);
    //System.out.println("this node is client:"+NameArray[u]);
    int HopCounter_d = 0;

    for(int k=0;k<n;k++)
     {
      FindClient= false;
      if(ConnArray[u][k] != 0 && FindClient == false)
       {
        String Str1=NameArray[ClientIndex[0]];
       //System.out.println(""+Str1);
       String Str2=NameArray[k];
       //System.out.println(""+Str2);

       if(Str1.equals(Str2))
        {
        int Value3 =0;
        Value3 = ConnArray[u][k];
        //System.out.println(""+Value3);
        FindClient= true;
        list1.add(Value3);
       }
       else
       {
         int Value4 = 0;
         Value4 = ConnArray[u][k];
         //System.out.println(""+Value4);
         HopCounter_d = HopCounter_d +1;
         SearchForCl(u,k,Value4,HopCounter_d,Hop*2);
       }
       }
       }
      }
      }
 Object clientcost[] = list1.toArray();
 list1.clear();
 for(int row =0;row<clientcost.length;row++)
  {
  System.out.print(clientcost[row]);
  ClientCost += ((Integer) clientcost[row]).intValue();
  System.out.println("");
  }
```

```
 System.out.println("the Transport Cost for Client "+Name+p +" is "+ClientCost);
 System.out.println("The Cpu Cost: "+CpuCost[ClientIndex[0]]);
 System.out.println("The Mem Cost: "+MemCost[ClientIndex[0]]);
 ClientCost = ClientCost + CpuCost[ClientIndex[0]]+ MemCost[ClientIndex[0]];
  System.out.println("the Total Cost for Client "+Name+ClientIndex[0] +" is
 "+ClientCost);
 CompareArray[p]= ClientCost;
 ClientCost =0;

 }
 }
System.out.println("");
/*for(int row =0;row<n;row++)
 {
  System.out.print(CompareArray[row]);
 }*/
System.out.println("");
int minX = Integer.MAX_VALUE ;

for(int d =0;d<n;d++)
     {
    if(CompareArray[d] !=0 && CompareArray[d]<minX)
     {
      minX = CompareArray[d];
      Index[0] = d;
     }
     }
 System.out.println("");
 System.out.println("ServerCost: "+ServerCost);
 System.out.println("ServerIndex[0]: "+ServerIndex[0]);
 //System.out.println("minX: "+minX);
 System.out.println("Index[0]: "+Index[0]);
 System.out.println("");

 if(minX > ServerCost)
   { System.out.println("Server "+Name+ServerIndex[0]+" remain at its Original place ");
   }
 else
  {
   System.out.println("");
   System.out.println("Server "+Name+ServerIndex[0]+" moves itself to Node
   "+Name+Index[0]);
    Charray[ServerIndex[0]] = "NonClient";
    Charray[Index[0]] = node.BecomeServer();
    System.out.println("Client "+Name+Index[0]+" Become "+Charray[Index[0]]);
     System.out.println("Server "+Name+ServerIndex[0]+" Become "
   +Charray[ServerIndex[0]]);
    System.out.println("");
    }

ServerCost = 0;
for(int row =0;row<n;row++)
 {Clients[row]=null; }
for(int row =0;row<n;row++)
 {CompareArray[row]=0;}
```

```
                }
                }
                }
public void shuffleArray(int[] a)
  {
   int length = a.length;
   Random random = new Random();
   random.nextInt();
   for(int i=0;i<n;i++)
   { int change = i + random.nextInt(n-i);
     Swap(a, i, change);}
  }
public void Swap(int[] a, int i, int change)
   {
     int x = a[i];
     a[i] = a[change];
     a[change] = x;
    }

public void SearchForSer(int x,int y, int p,int z, int t)
     {
       for(int s=0; s<n ; s++)
       {
      if(ConnArray[y][s] != 0 && s!=x && FindServer == false)
       {
        HopArray_a[s] = z + 1;
        int HopCounter_c = 0;
        HopCounter_c = HopArray_a[s];
        int Value5 =0;
        int Value6 =0;

        String Str1=NameArray[ServerIndex[0]];
        //System.out.println(""+Str1);
        String Str2=NameArray[s];
        //System.out.println(""+Str2);
        if(Str1.equals(Str2))
         {  //System.out.println(""+ConnArray[y][s]);
           Value5 = ConnArray[y][s]+p;
           //System.out.println(""+Value5);
           FindServer= true;
           list.add(Value5);
         }
         else
         {
          if(HopCounter_c<t)
           {
            Value6 = ConnArray[y][s]+p;
            SearchForSer(y,s,Value6,HopCounter_c ,t);
           }
           }
           }
           }
           }
```

```
public void SearchForCl(int x,int y, int p,int z, int t)
    {
     for(int s=0; s<n ; s++)
      {
      if(ConnArray[y][s] != 0 && s!=x && FindClient ==false)
       {
        HopArray_b[s] = z + 1;
        int HopCounter_c = 0;
        HopCounter_c = HopArray_b[s];
        int Value7 =0;
        int Value8 =0;
        String Str1=NameArray[ClientIndex[0]];
        //System.out.println(""+Str1);
        String Str2=NameArray[s];
        //System.out.println(""+Str2);

        if(Str1.equals(Str2))
         {  //System.out.println(""+ConnArray[y][s]);
           Value7 = ConnArray[y][s]+p;
           //System.out.println(""+Value7);
           FindClient = true;
           list1.add(Value7);
         }
         else
         {
          if(HopCounter_c<t)
           {
           Value8 = ConnArray[y][s]+p;
           SearchForCl(y,s,Value8,HopCounter_c ,t);
           }
           }
           }
           }
           }
public void Searchfor(int x,int y,int z,int t)
    {
     for(int s=0; s<n ; s++)
      {
      if(ConnArray[y][s] != 0 && s!=x && Charray[s].compareTo("Server") != 0)
       {HopArray[s] = z + 1;
        int HopCounter_a =  HopArray[s];
        if(Charray[s].compareTo("Client")==0)
       {Clients[s]=(""+Name+s);
        if(HopCounter_a<t)
        {Searchfor(y,s,HopCounter_a,t); }
        else {
        if(HopCounter_a<t)
        {Searchfor(y,s,HopCounter_a,t); }
        }
        }
        }
        }
        }
```

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die gegebenen Quellen benutzt habe. Zitate oder sinngemäße Wiedergaben von Inhalten dieser Literatur wurden gekennzeichnet. Diese Arbeit wurde noch nicht an anderer Stelle für Prüfungszwecke vorgelegt.

Ilmenau, den 14.03.2011

Imad Kailouh